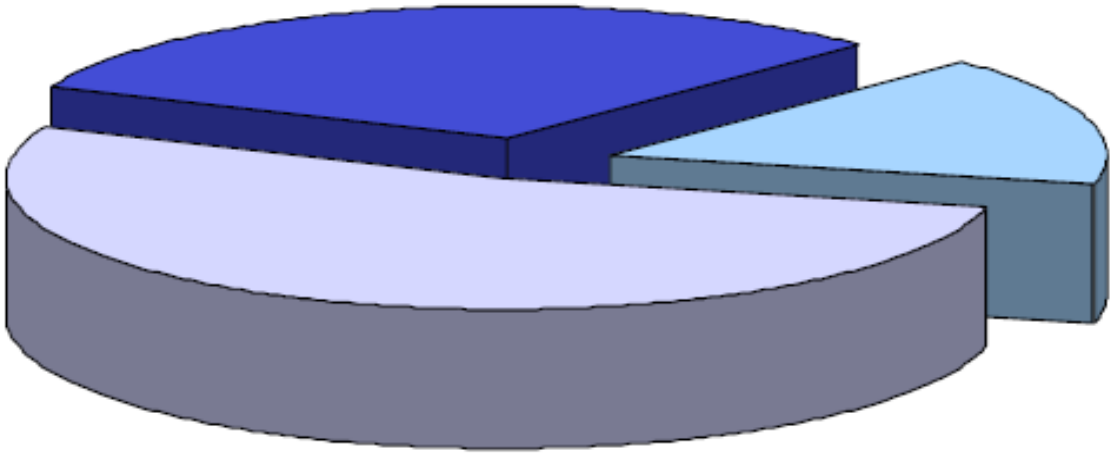
Качеството на разработваното ПО

Софтуерни проекти

31 %
прекратени



16 %
успешно
завършили

51 %
спорни

Провалени проекти

- Бюджет

под 20 %	15.5%
21-50 %	31.5 %
51-100 %	29.6 %
101-200 %	10.2 %
201-400 %	8.8 %
над 400 %	4.4 %

Средната себестойност на провалените е 189 % от предвидената

Провалени проекти

- Продължителност

под 20 %	13.9 %
21-50 %	18.3 %
51-100 %	20.0 %
101-200 %	35.5 %
201-400 %	11.2 %
над 400 %	1.1 %

Средната продължителност на провалените е 222 % от предвидената

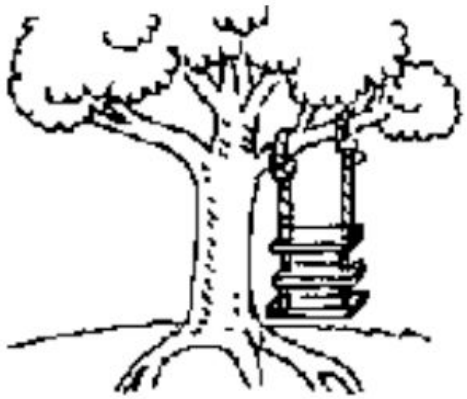
Провалени проекти

- Реализирана функционалност

под 25 %	4.6 %
25-49 %	27.2 %
50-74 %	21.8 %
75-99 %	39.1 %
100 %	7.3 %

Средното изпълнение при спорните проекти е 61 % от заданието.

Основен проблем



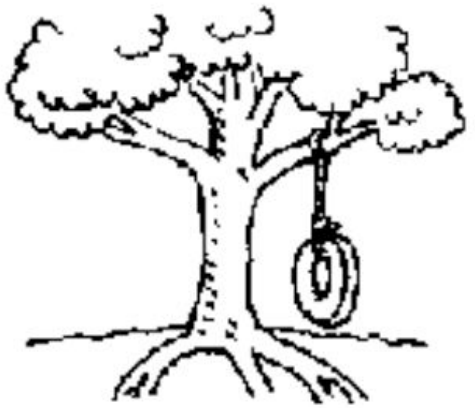
Това е разбрано, че ще се разработва

Задание в проекта

Програмен дизайн

Разработка

Инсталирана версия



<u>Желание на клиента</u>

Основен проблем

- По малко от половината ръководители на проекти смятат, че качеството на програмните продукти се подобрява.

Значително повече повреди (failures)	27%
Повече повреди (failures)	21 %
Без промяна	11 %
По-малко повреди (failures)	19 %
Значително по-малко повреди (failures)	22 %

Качество на ПО

- Под качество на ПО хората разбират две неща:
  - ✓ Програмният продукт трябва да прави това, за което е предназначен и нищо друго (“трябва да прави правилните неща”)
  - ✓ При разработка да извършва дейностите си коректно и качествено (“трябва да работи правилно”).

Качество на ПО

- Дейности по за създаване на качествено ПО:
  - ✓ Планиране на качеството
  - ✓ Изпълнение на дейностите по валидация и/или верификация на разработваното ПО
  - ✓ Измерване и анализ на работата на ПО за установяване на данни, показващи съответното ниво на качество

Стремеж към все по-големи ПС

- Това са системи с поне 1 млн. реда и имащи висока сложност:
  - ✓ Разработват се голям брой хора.
  - ✓ Разработката е в дълъг период от време.
- Основни проблеми
  - ✓ Безразборно добавяне на функционалност, която често е несъществена.
  - ✓ Лош и даже ‘глупав’ дизайн.
  - ✓ Неправилен избор на език за програмиране.
  - ✓ Несъстоятелен подбор на технологии за разработка.

Подобряване на качеството

- Дейностите за подобряване на качеството на разработката са насочени към
  - ✓ Предпазва от появата на някои класове проблеми/повреди.
  - ✓ Премахва възможността от възникване на някои класове проблеми/повреди.
  - ✓ Намалява вероятността и прави по-трудна появата на проблеми/проблеми с използваното ПО.

Управление на дейностите за подобряване на качеството

Качество на ПО

- Алтернативният въпрос е:
  - ✓ “Какви са характеристиките на висококачественото ПО ?”
- За дефинирането на тези храктеристики е необходимо да се отчитат различни гледни точки и очаквания:
  - ✓ Потребители
  - ✓ Разработчици
  - ✓ Ръководители на проекти
  - ✓ Специалисти по маркетинг на ПО
  - ✓ Мениджъри на софтуерни фирми
  - ✓ Специалисти по поддръжка и съпровождане на ПО

Качество на ПО

- Прието е да се използват пет основни гледни точки, дефиниращи характеристиките за качество:
  - ✓ Абстрактни критерии
  - ✓ Потребителско разбиране
  - ✓ Производствено разбиране
  - ✓ Продуктова същност
  - ✓ Стойностно-базирани критерии

Качество на ПО

- Прието е да се използват пет основни гледни точки, дефиниращи характеристиките за качество:
  - ✓ Абстрактни критерии
    - Трудни за описание характеристики, които правят потребителите доволни и щастливи
  - ✓ Потребителско разбиране
  - ✓ Производствено разбиране
  - ✓ Продуктова същност
  - ✓ Стойностно-базирани критерии

Качество на ПО

- Прието е да се използват пет основни гледни точки, дефиниращи характеристиките за качество:
  - ✓ Абстрактни критерии
  - ✓ Потребителско разбиране
    - Дали се реализира това, което потребителите очакват
  - ✓ Производствено разбиране
  - ✓ Продуктова същност
  - ✓ Стойностно-базирани критерии

Качество на ПО

- Прието е да се използват пет основни гледни точки, дефиниращи характеристиките за качество:
  - ✓ Абстрактни критерии
  - ✓ Потребителско разбиране
  - ✓ Производствено разбиране
    - Дали са спазени определените производствени стандарти
  - ✓ Продуктова същност
  - ✓ Стойностно-базирани критерии

Качество на ПО

- Прието е да се използват пет основни гледни точки, дефиниращи характеристиките за качество:
  - ✓ Абстрактни критерии
  - ✓ Потребителско разбиране
  - ✓ Производствено разбиране
  - ✓ Продуктова същност
    - Дали са спазени определените производствени стандарти
  - ✓ Стойностно-базирани критерии

Качество на ПО

- Прието е да се използват пет основни гледни точки, дефиниращи характеристиките за качество:
  - ✓ Абстрактни критерии
  - ✓ Потребителско разбиране
  - ✓ Производствено разбиране
  - ✓ Продуктова същност
  - ✓ Стойностно-базирани критерии
    - Определят готовността на потребителите да платят за разработения продукт

Качество на ПО

- Основни групи според ролите си по отношение на разработваното ПО :
  - ✓ Клиенти
    - Купувачи (customer) –избират какво и защо да се купи
    - Потребители (user) – само използват вече закупено/осигурено ПО
      - В тази група влизат и ‘нехора’-потребители, т.е. друго ПО или апаратна част, взаимодействащи с разработваното ПО
  - ✓ Производители (producers)
    - Директно участващите в разработката (дизайнери, програмисти, ръководители и др.)
    - Хората от трети фирми, подпомагащи разработката, поддръжката и съпровождането на ПО

Клиенти и Качество

- Основното очакване е ПО да реализира в действителност полезните функции, които казва че реализира:
  - ✓ Функциите отговарят на нуждите на покупателите
  - ✓ Издръжливост на ПО (reliability)
    - При многократна употреба и/или в дълъг период от време се запазва полезността и правилното функциониране на ПО.
- Специфирни изисквания на купувачи
  - ✓ цена

Клиенти и Качество

- Специфични изисквания на потребителите
  - ‘Използваемост’ (usability)
    - Сложността на използването може да е най-важна храктеристика за много потребители
  - Сложността на инсталирането и поддръжката е друга важна характеристика за много хора
  - Адаптивността (adaptability)
    - Важно е както за хората-, така и за ‘нехората’-потребители
  - Възможността за взаимодействие в една по-голяма система (inter-operability)
    - Особено важно за групата на ‘нехората’-потребители

Производители и Качество

- Основната оценка е:
  - ✓ Дали разработеното ПО е това, което е поръчано, т.е. дали изпълнява всички изисквания, заложени в спецификацията на проекта.
  - ✓ Колкото по-лесно това се доказва, толкова по-качествено е разработено ПО.
- Допълнително изискване:
  - ✓ Купувачите пак да се върнат при нас.
  - ✓ Да мога да поема друг проект след завършването на текущия.

ISO-9126

- Стандарт за качество на ПО, дефиниращ 6 характеристики за качество:
  - ✓ Функционални:
    - годност (suitability), точност на разработката (accuracy), съвместимост(interoperability), сигурност (security)
  - ✓ Надеждност:
    - работоспособност (maturity), устойчивост на откази (fault tolerance), възстановимост (recoverability)
  - ✓ Използваемост:
    - разбираемост на ПО (understandability), научаемост (learnability), леснота на опериране със ПО (operability)

ISO-9126

- Стандарт за качество на ПО, дефиниращ 6 характеристики за качество:
  - ✓ Ефективност:
    - измерва връзка ‘функионалност-бързодействие-ресурси’
  - ✓ Възможност за поддръжка (maintainability):
    - анализуемост, променяемост (changeability), устойчивост (stability), възможност за тестване на промените (testability)
  - ✓ Преносимост:
    - адаптивност, инсталируемост, съответствие между инсталираните версии (conformance), заменяемост на версиите (replaceability)

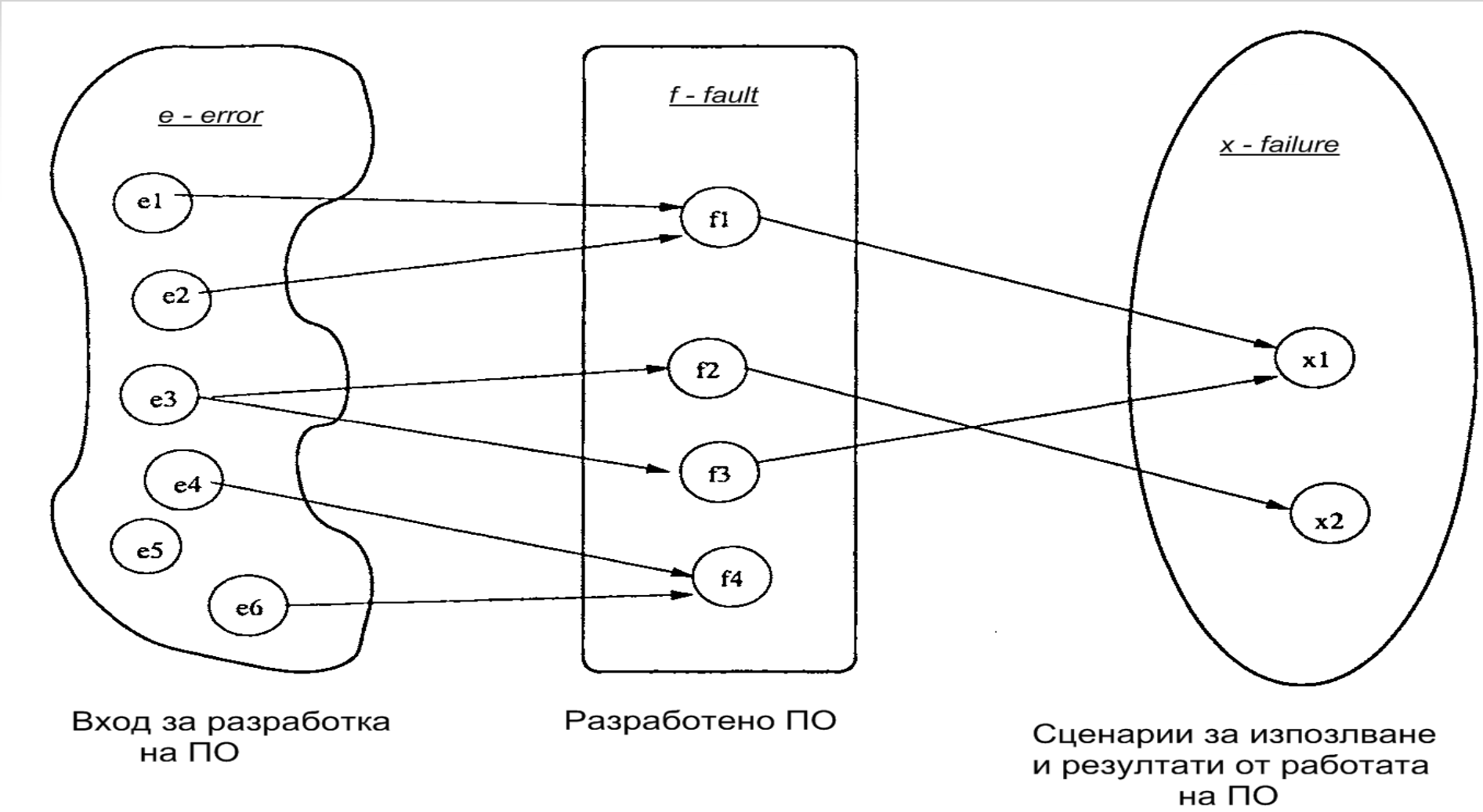
Други стандарти за качество

- CUPRIMDS
  - ✓ Capability, Usability, Performance, Reliability, Installation, Maintenance, Documentation, Service
  - ✓ Използва се в IBM
- Web-базирани приложения
  - ✓ Първични характеристики: издръжливост, използваемост и сигурност
  - ✓ Вторични характеристики: достъпност (availability), мащабируемост (scalability), възможност за поддръжка и времето за излизане на пазара
- Ежедневно използвано ПО
  - ✓ Основните критерии в този случай са коректността и съответствие със заданието

Дефекти на ПО (defects)

- Повреда (failure)
  - ✓ Невъзможност на система или компонент да реализират заложена функционалност според зададени технически характеристики
  - ✓ Отнася се за отклонение в поведението на ПО спрямо заданието за разработка
- Дефект/опущение (fault)
  - ✓ Неправилен процес, стъпка или дефиниция на данни в компютърна програма.
  - ✓ Отнася се към наличието на специфични условия в ПО, позволяващи проявлението на дадена повреда.
- Грешка (error)
  - ✓ Човешка дейност, водеща до получаване на некоректни резултати.
  - ✓ Отнася се до отсъстващи или неправилни човешки действия, водещи до появата/залагането на дефекти в ПО.

Недостатъци на ПО (defects)



Коректността като основа на качеството

- Дефиниране
  - От гледна точка на клиентите е свързана с външното проявление на ПО, т.е. наличието на повреди.
    - Оказва влияние върху използваемостта, преносимостта, възможността за инсталиране, поддръжката и т.н.
  - От гледна точка на производителите е свързана с влиянието на вътрешните особености на ПО, т.е. наличието на дефекти.
    - Свързана е характеристики като дизайн, размер, сложност и т.н.

Клиентите и Коректността

- Коректността може да се измерва чрез различни техники за следене на повредите:
  - ✓ Описват се типа и същността на проявлението
  - ✓ Измерват се чрез анализ на броя, разпространението, честотата на проявление.
- Издръжливостта (reliability) пряко зависи от коректността, по-точно от честотата на проявление на повреди
  - ✓ Дефинира се като време за работа без поява на повреди или като работа при определени входни данни без проява на повреди
- Безопасността на ПО (safety) също зависи от коректността
  - ✓ Дефинира се като характеристика на ПО, което не води до инциденти при определено ниво на опасност от настъпилата повреда.
    - × Инцидентът е повреда в ПО с тежки последствия за клиента.

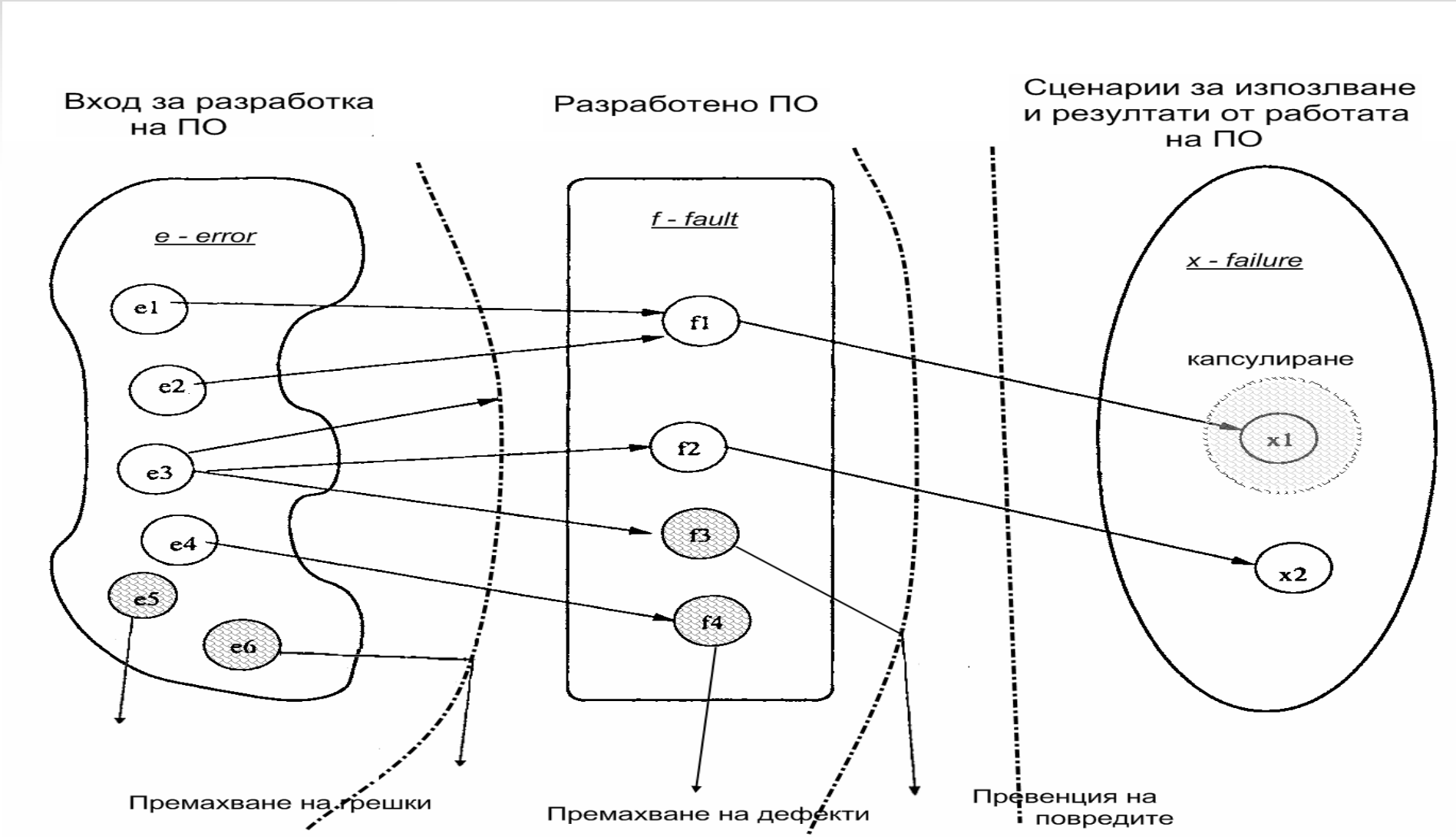
Производители и Коректността

- Използват се техники и методи за измерването и анализ на дефектите в разработката на ПО:
  - ✓ Ниво индивидуални анализ:
    - Тип, връзка с определени повреди и инциденти, причини, време и причини за тяхната поява в ПО и др.
  - ✓ Ниво на групов анализ
    - Изследват се разпространението и гъстотата на дефектите в различните фази на разработка.
    - Изследват се разпространението и гъстотата на дефектите в различните подсистеми и/или компоненти на ПО.
- Разработват се специални планове, съдържащи мерки по фази и компоненти за подобряване на коректността, т.е. на качеството на ПО.

Сигурност в качеството на разработката на ПО

- Сигурността в качеството на разработката на ПО (quality assurance - QA):
  - ✓ Дефинира се като ‘Отношение към наличието на дефекти в ПО и техниките за тяхното ограничаване/премахване’.
- Три генерични категории на проявление:
  - ✓ Превенция на дефектите чрез блокиране на грешките или тяхното премахване
  - ✓ Редуциране на дефектите чрез разпознаване на дефектите и последващото им премахване
  - ✓ Ограничаване на влиянието на повредите чрез превенция и капсулиране (ограничаване на проявленията)

Сигурност в качеството на разработката на ПО



Превенция на дефектите

- Обучение на хората
  - ✓ Много източници на грешки са резултат от незнание или неразбиране
- Формални методи
  - ✓ Осигуряват определянето на местата с лош дизайн и/или реализация
- Използването на стандарти и добри практики при разработката
  - ✓ Силно се ограничавам възможностите и местата за вмъкване (инжектиране) на грешки в разработваното ПО.
- Използване на подходящи и сигурни средства за разработка
  - ✓ Намалява се възможността за добавяне на грешки от работата на използваното ПО при разработката.

Редукция на дефектите

- Инспекция
  - ✓ За директно откриване и премахване на дефекти
- Тестване
  - ✓ За наблюдение на повредите като начин за намиране и премахване на дефектите
- Други методи
  - ✓ Това са методи за анализ на ПО след създаването му

Инспекция

- Това е статичен метод за анализ, базиращ се на:
  - ✓ Критично четене на различните документи по създаване на ПО (изисквания, дизайн, код, т.н.).
  - ✓ Реализира се на последователни и координирани сесии от екип от специално обучени специалисти, работещи координирано.
  - ✓ Дефектите се определят директно при провежданите инспекции
  - ✓ Откритите грешки се премахват и се извършва повторна инспекция за поява на нови.
  - ✓ Структурата и същността на процеса силно варират от обикновено преминаване през документите до формализиране и анализ на инспектирания документ.
  - ✓ Процесът задължително се планира като дейности и насоки.

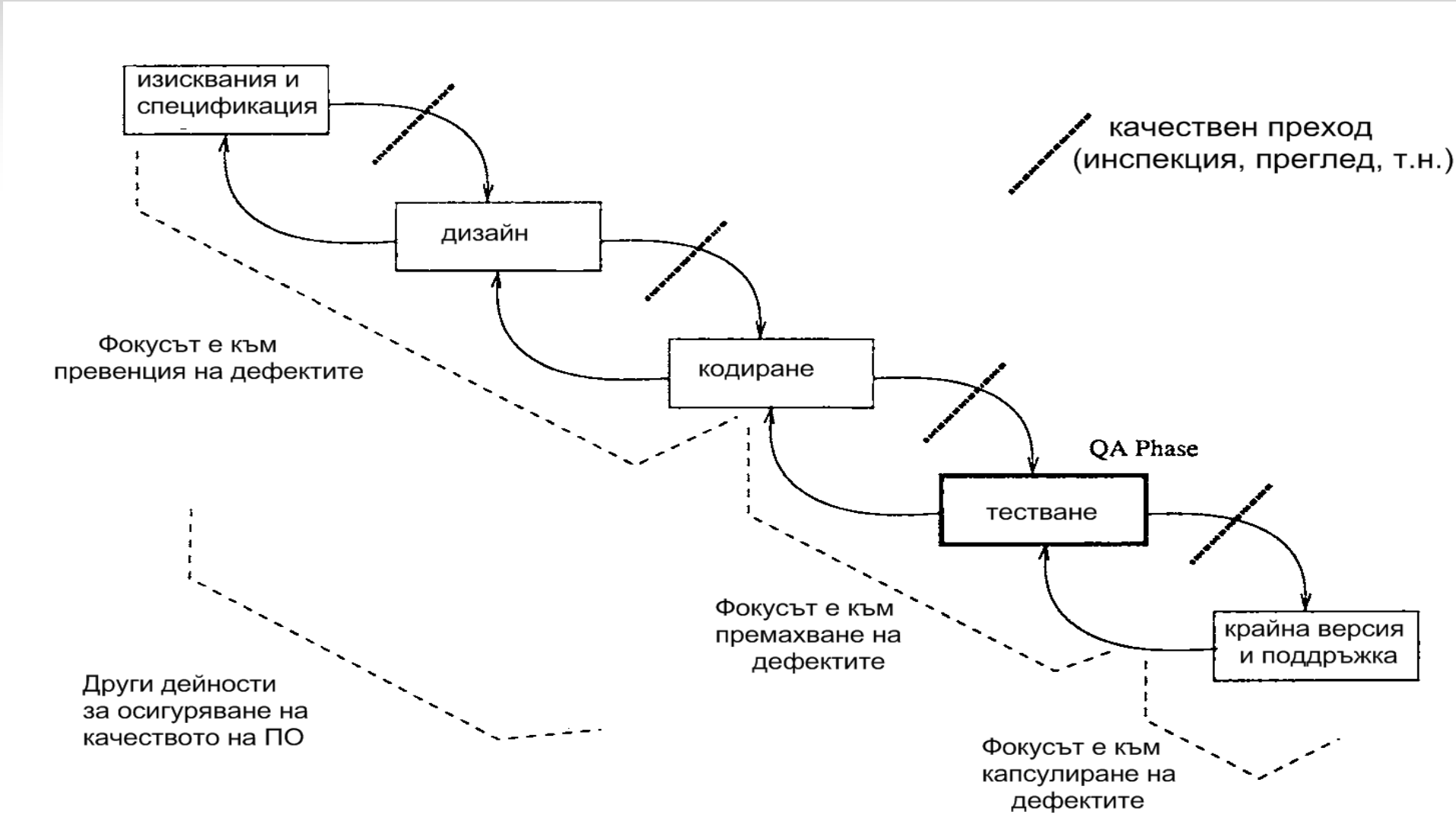
Тестване

- Кога дефекти трябва да се определят чрез тестване ?
- Какво да се тества и какъв вид дефекти се намират ?
- При какво ниво на открити дефекти да се спира тестването ?

Осигуряване на качеството в процесите на разработка

- Управление на дефектите (defect handling)
  - ✓ Документиране на проявата и начално описание на дефекта (defect logging)
  - ✓ Следене и описание на проявленията на дефекта след началното установяване до крайното премахване (defect tracking)
- Дейности по управление на дефектите
  - ✓ Анализ на дефектите – дефекти се обработват по групи (вътрешни грешки, грешки на потребителя, други) за да има консистентност по типовете грешки
  - ✓ Периодично наблюдение и описание на състоянието на всички установени дефекти.

Пример


изисквания и спецификация
дизайн
кодиране
тестване
крайна версия и поддръжка
качествен преход (инспекция, преглед, т.н.)
Фокусът е към превенция на дефектите
QA Phase
Фокусът е към премахване на дефектите
Други дейности за осигуряване на качеството на ПО
Фокусът е към капсулиране на дефектите

Валидация и качеството на ПО

- Валидацията е насочена към установяване на съответствието между функционалността на ПО и очакването на клиентите
  - ✓ Проверява дали е реализирана функционалността
  - ✓ Проверява дали има отклонение в поведението на ПО спрямо описаното в заданието
  - ✓ Проверява как повредите влияят върху възможността за използване на заложената функционалност.

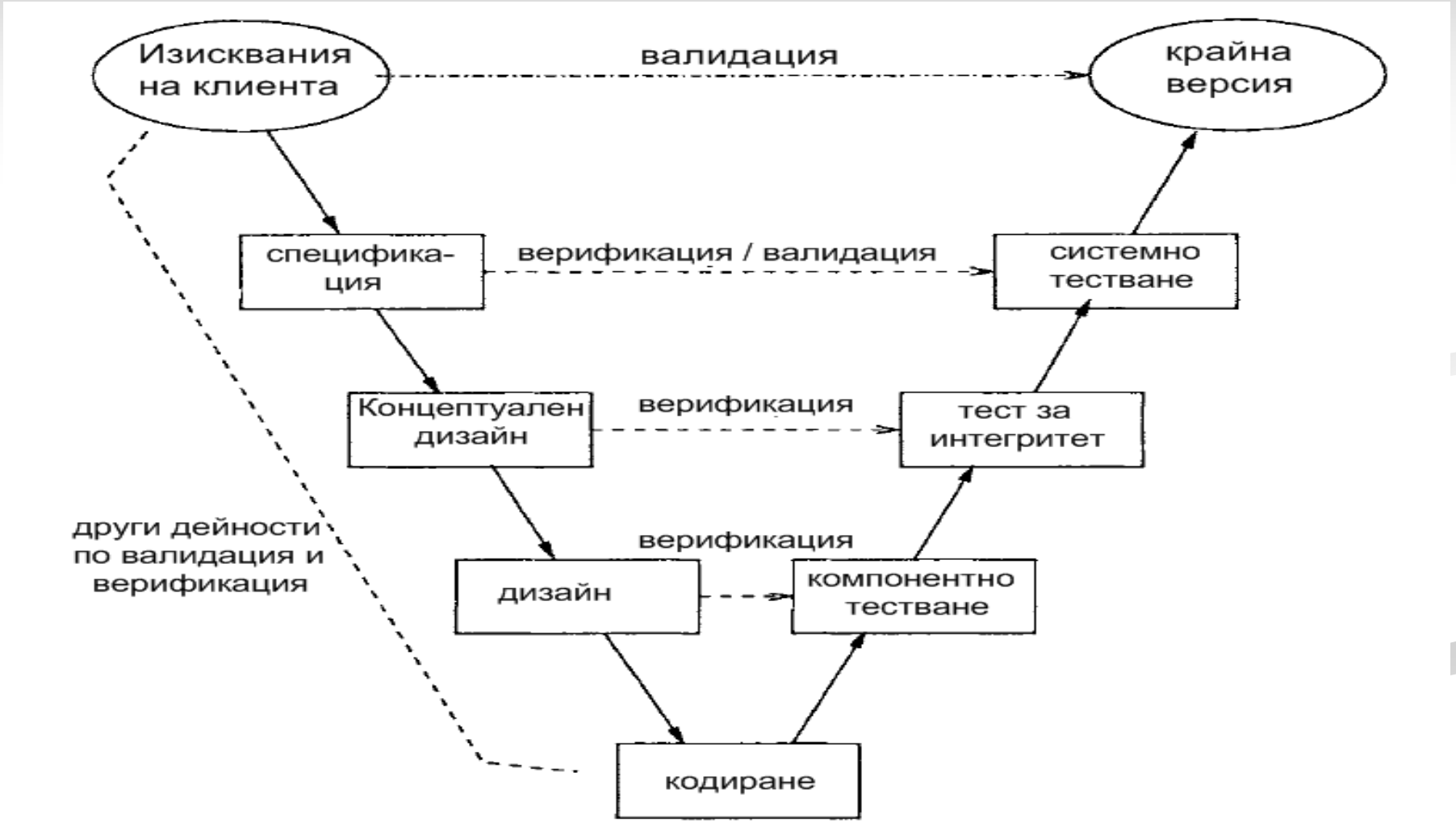
Валидация и качеството на ПО

- Валидационни дейности, отговарящи за качеството на ПО
  - ✓ Системно тестване (system testing)
  - ✓ Тестване за допустимост (acceptance testing)
  - ✓ Статистически анализ на използваемостта на заложената функционалност
  - ✓ Установяване на дефектоустойчивост (fault tolerance).
  - ✓ Установяване на сигурността на ПО (safety assurance)

Верификация и качеството на ПО

- Верификацията проверяват дали реализацията е извършена по зададените спецификации
  - ✓ При наличието на повреди се проверяват причините за изменения в поведението на ПО
  - ✓ При други видове проблеми се проверява за грешки и вкарани дефекти.

Валидация и Верификация на ПО като част от качеството на разработката



Questions ?

Основни принципи на валидация и верификация на програмно осигуряване

Методи и стратегии на реализация

Възникване на V&V

- Валидацията и верификацията (V&V) са резултат от развитие на дейностите по осигуряване и управление на качеството на разработвания продукт.
  - ✓ Развитието на системите и методите за управление на процесите на управление на разработката периодично водят до промяна на основната насока и задачи на дейностите по V&V.
  - ✓ Появата на нови приложения на ПО води и до необходимостта от разширяване (по-рядко предефиниране) на тези понятия.
  - ✓ Това води до съществуващата неоднозначност в разбирането за тези процеси.

Понятието ‘Система’

- Този термин възниква в 20-и век като резултат от развитието на теорията за системите на Bertalanffy.
- Интуитивно разбиране
  - ✓ Тя се състои от отделни елементи, организирани по някакъв начин в едно цяло, имащи зададена интерфейс помежду си.
  - ✓ Работата на системата създава резултат, който не може да се получи като елементарно сумиране на резултатите от работата на отделните елементи.
  - ✓ Всяка система влияе на заобикалящата я среда, както и се влияе от тази среда – задължително се дефинира връзката между системата и заобикалящата я среда
    - Формите на вход-изход могат да бъдат най-различни (материя, енергия, информация).

Инженерна система

- Целта на инженерния процес е разработката на ефективни и надеждни системи (продукти, услуги, решения), отговарящи на определени нужди, които се дефинират чрез задаване на множество от ограничения.
- Класическият цикъл на живот за всяка инженерна система включва следните фаза: дефиниране, дизайн, реализиране, интеграция, окачествяване и производство.

Инженерна система

- Основни характеристики
  - ✓ Търсене на единично решение, базиране на уникален дизайн и ориентирано към специфичен проблем.
  - ✓ Поведението на системата трябва да е предвидимо и да бъде прецизно описано.
  - ✓ Подходът за реализация е ‘отгоре-надолу’, базиран на фундаменталното разбиране, че всяка система може да бъде описана цялостно чрез описание на поведението на нейните части, връзките им и взаимното им влияние.

Инженерна система

- Основни атрибути:
  - ✓ Предсказуемост: във всяка ситуация системата работи по предсказуем начин.
  - ✓ Надеждност: в зададен период от време и при зададени условия на работа системата изпълнява изискваните дейности.
  - ✓ Прозрачност: структурата на системата и на процесите и може да се опише експлицитно.
  - ✓ Управляемост: системата може да се управлява директно според зададените инструкции при зададените условия на работа.

Инженерна дефиниция за система

“Система е множество от различни елементи, които заедно създават резултат, които не може да се получи от самостоятелните елементи. Елементи (части на системата) могат да включват хора, апаратура, програмно осигуряване, средства, политики, документи. И всички тези елементи са необходими за получаване на необходимите резултати. Тези резултати включват: свойства, характеристики, функции, поведение, действия. Ползата от системата, като цяло, която се получава извън полезността на всеки от елементите поотделно, основно се получава от връзката между отделните части, т.е. от начина на свързването им.”

INCOSE

V&V – основна цел

- Основна дилема

“Елиминиране на дефектите

или

Доказване, че няма дефекти”

V&V – основна цел

- Елиминиране на дефектите
  - ✓ Необходимо е да се премахнат колкото е възможно повече дефекти при ограничението за пари, време, ресурси (апаратура, хора, т.н.).
  - ✓ Основен подход не само за ПО, но и за всички инженерни проекти - позволява да се оцени и подобри качеството на продукта.
- Доказване, че няма дефекти
  - ✓ Много необходимо от маркетингова гледна точка, но неправилно от инженерна.
  - ✓ В болшинството от случаите е невъзможна за реализация – изключителни обеми от работа с висока сложност и необходимост от безкрайни ресурси.

Атрибути на качеството

- За оценка на процеса на V&V се използват следните атрибути:
  - Коректност (correctness)
    - Ако входните данни са валидни, ПО ще работи ли както се очаква ?
  - Пълното (completeness)
    - Реализира ли ПО всички изисквания, поставени за него ?
  - Консистентност (consistency)
    - По еднакъв начин ли работи ПО в идентични ситуации ? Аналогичното ПО от дадената фамилия също ли има този начин на работа в тези ситуации ?
  - Надеждност (reliability)
    - Във всички случаи ли системата има очакваното поведение, даже и при силно променени условия ?

Атрибути на качеството

- За оценка на процеса на V&V се използват следните атрибути:
  - Полезност (usefulness)
    - ПО дава ли на потребителите полезни услуги ?
  - Използваемост (usability)
    - Удобна ли е системата за употреба ?
  - Ефективност (efficiency)
    - Ефективно ли използва програмното осигуряване ресурси като време, памет, периферия, комуникационни канали и други ?
  - Сполучливост (effectiveness)
    - От гледна точка на вложениете пари това едно ефективно решение ли е ?

Същност на V&V

- Това са дейности, които трябва да се извършват се през целия жизнен цикъл на ПО за да позволят дефектите да се установяват колкото може по-рано.
- Дейностите трябва да се планират, документират и ръководят от безпристрастна група специалисти.
- Проверката на отделните части на ПО не гарантира задължително качеството на ПО в цялост. Цялостната проверка на системата е много сложен процес и не винаги може да се реализира.

Същност на V&V

- Валидация
  - ✓ Използва се за сравняване на системата от изисквания на клиентите (stakeholders), които могат да се променят във времето.
  - ✓ За реализацията и е необходимо наличие на формално или неформално специфициране на множеството от желания на клиентите за функционалност на разработваното ПО преди началото на процеса.
- Верификация
- Тестване

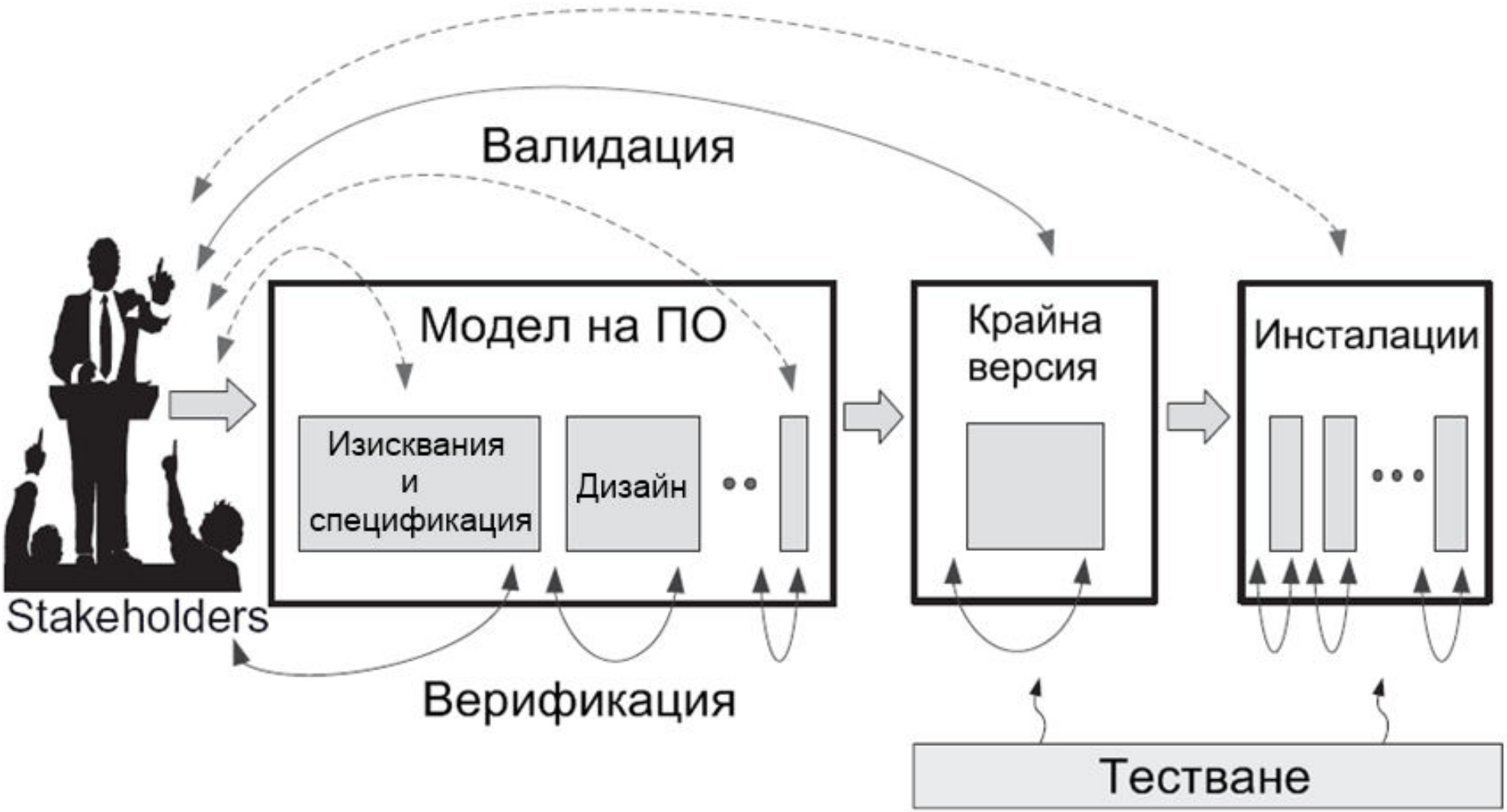
Същност на V&V

- Валидация
- Верификация
  - ✓ Има отношение към писаните спецификации на системата
  - ✓ Включва коректността на вътрешната структура на ПО
  - ✓ Свързана е със процесите на разработка от целия цикъл на живот за ПО
- Тестване

Същност на V&V

- Валидация
- Верификация
- Тестване
  - ✓ Включва някои видове проверки на ПО – това са статични или динамични методи за оценка на коректността на функциониране.
  - ✓ Извършва се като подмножество на процесите на валидация или верификация.

Същност на V&V



Валидация

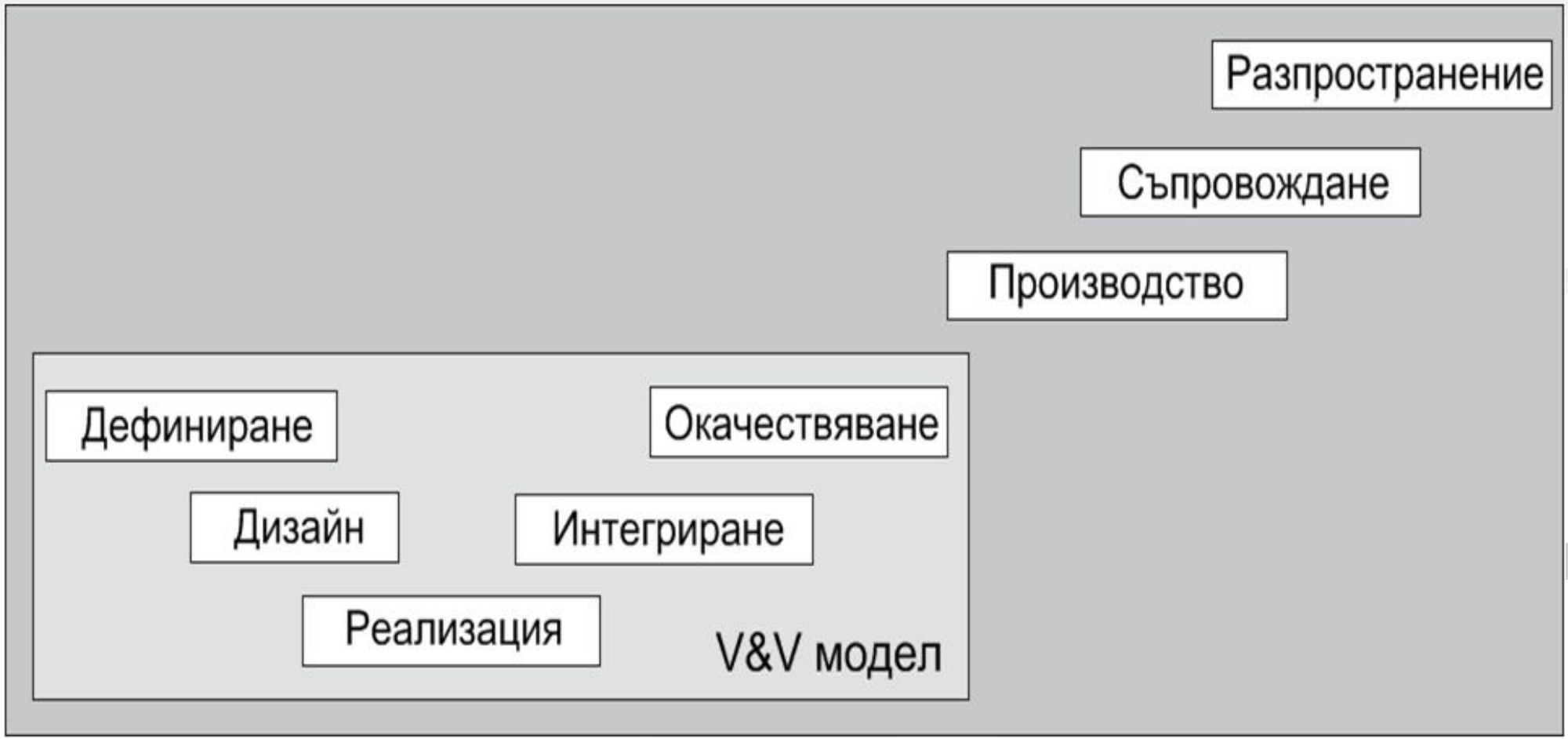
- Основни дефиниции
  - IEEE-610:
    - Процес на оценка на система или компонент по време на разработка или в края на разработката дали отговаря на специфицираните изисквания.
  - ANSI/EIA 632
    - Потвърждение чрез проучване и на основата на обективни данни, че зададената употреба на продукта (разработен или закупен) или съвкупността от продукти е извършено в желаната среда.

Верификация

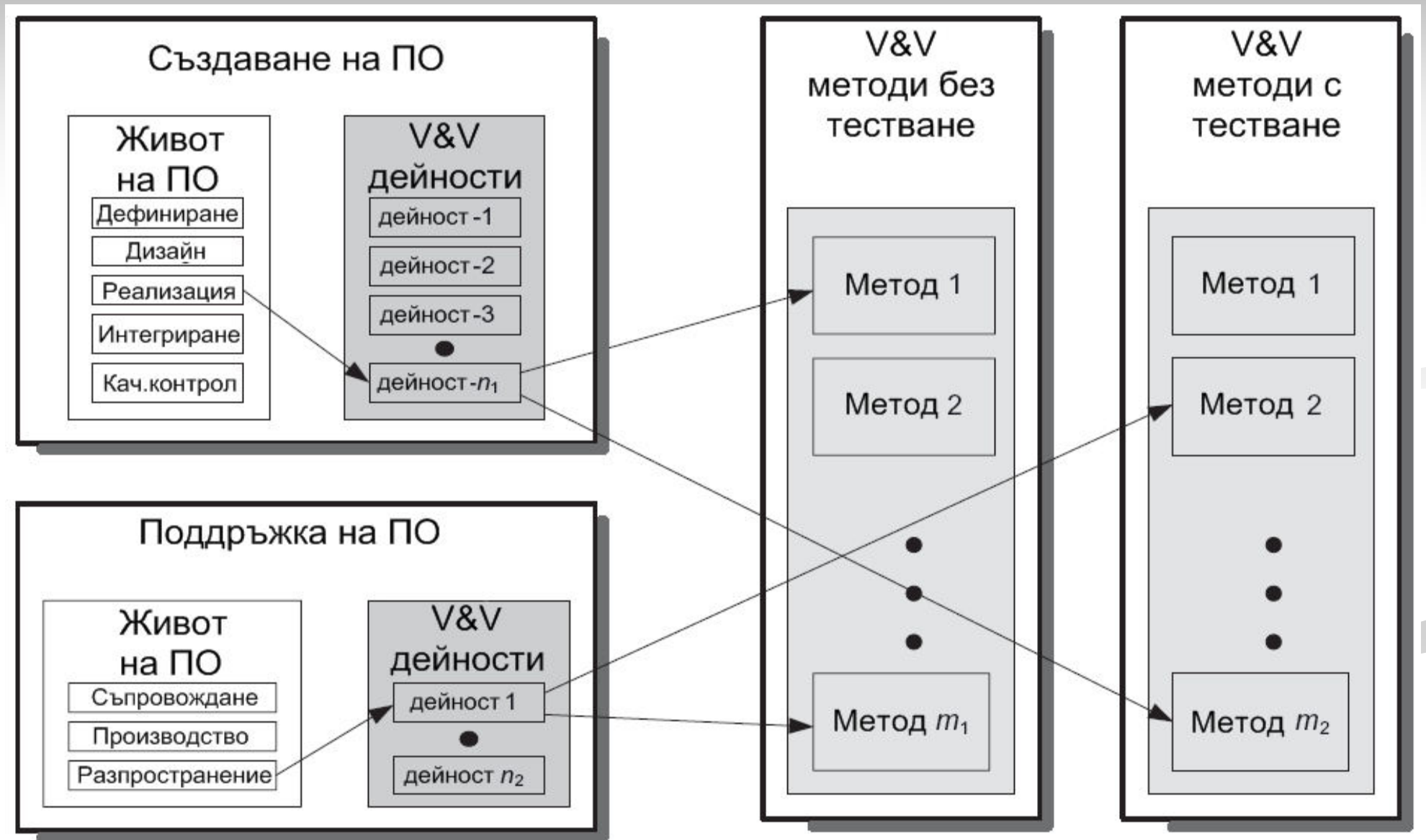
- Основни дефиниции
  - ✓ IEEE-610:
    - Процес на оценка на система или компонент за определяне дали продуктът в дадената фаза на разработка отговаря на условията, определени в началото на фазата.
  - ✓ ANSI/EIA 632
    - Потвърждение чрез проучване и на основата на обективни данни, че специфицираните изисквания, по които продуктът е създаден, кодиран или сглобен, са изпълнени.

Дали създаването на продукта (написан, разработен, кодиран, сглобен и интегриран) е извършено коректно?

Място на V&V модела



Реализация на V&V модела



Реализация на V&V модела

- Фраза “Дефиниране”
  - ✓ Определят се изискванията към разработваното ПО.
- Дейности за реализация на V&V модела
  - ✓ Моделът е насочен е към дефиниране на ясни, пълни, непротиворечащи си изисквания.
  - ✓ Дефинират се правилата за измерване на качеството.
  - ✓ Създава се план за окачествяване на ПО на основата на дефинираните изисквания.

Реализация на V&V модела

- Фраза “Дизайн”
  - ✓ Определя се техническата концепция и програмната архитектура за реализация.
- Дейности за реализация на V&V модела
  - ✓ Обвързване на изискванията от заданието с конкретни подсистеми и компоненти.
  - ✓ Конкретизиране на начините за измерване на качеството.
  - ✓ Избиране на стратегиите за верификация.
  - ✓ Дефиниране на изискванията за качество за следващите фази, които зависят от избрания дизайн.

Реализация на V&V модела

- Фраза “Реализация”
  - ✓ Реализация на дизайна в реално система.
- Дейности за реализация на V&V модела
  - ✓ Практическа верификация на ниво компоненти и подсистеми според ограниченията и изискванията, наложени от дизайна на ПО.
  - ✓ Генериране на предложения за подобряване на реалзиацията на отделни елементи на ПО.
  - ✓ Валидиране на основните изисквания според обвързването им с конкретни подсистеми и/или компоненти.

Реализация на V&V модела

- Фраза “Интегриране”
  - ✓ Свързване на отделните подсистеми и компоненти в едно цяло.
- Дейности за реализация на V&V модела
  - ✓ Верификация на интерфейса между отделните елементи.
  - ✓ Верификация на интерфейса между системата и обкръжаващата я среда.
  - ✓ Валидиране на изисквания, които са резултат от интеграцията на няколко подсистеми и/или компоненти.
  - ✓ Завършване на плана за окачествяване.

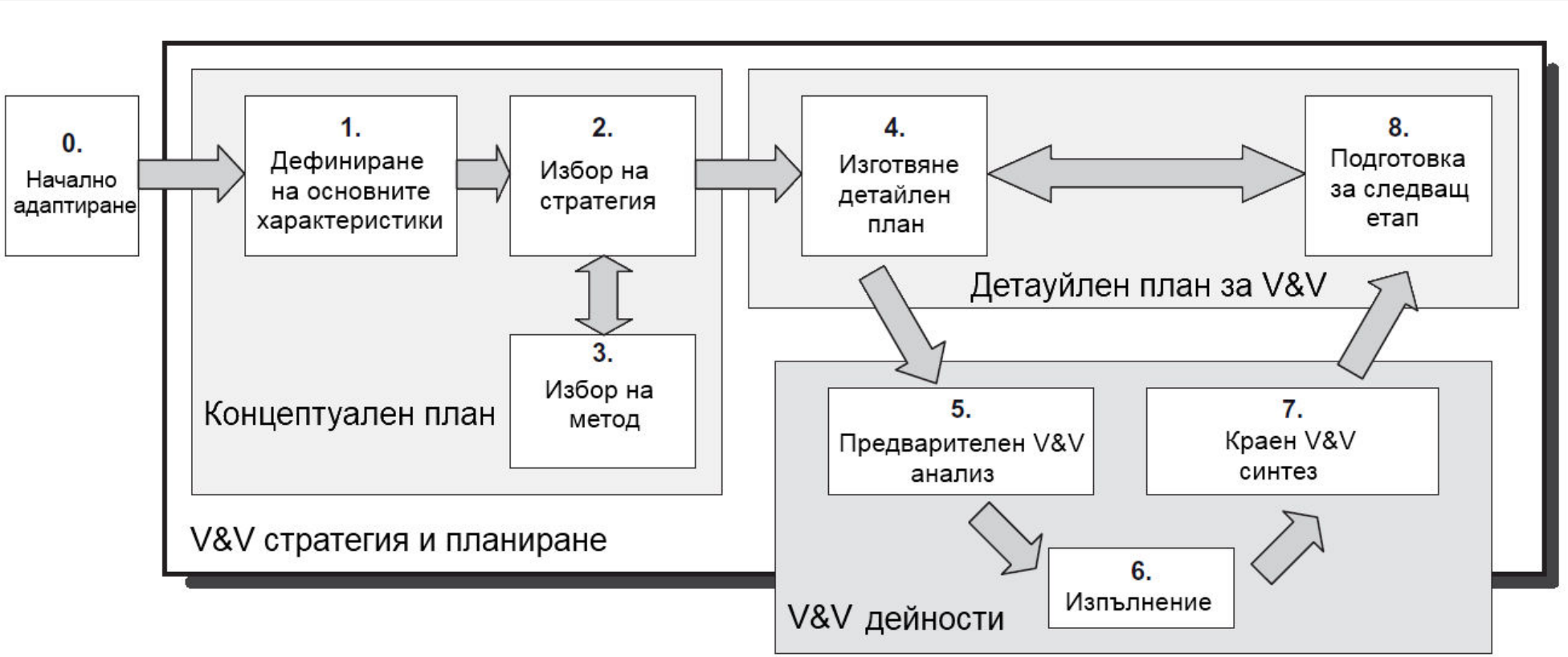
Реализация на V&V модела

- Фраза “Окачествяване”
  - ✓ Извършва се оценка на качеството чрез прилагане на тестове, специфицирани в стандарти, добри практики или определени от възложителя.
- Дейности за реализация на V&V модела
  - ✓ Валидация на верифицираните изисквания с цел проверка дали правилно са отразени и реализирани реалните изисквания на възложителя.
  - ✓ Проверяват се за правилност избраните методи за верификация (анализ, тестване, инспекция, прототипна демонстрация или други).

Реализация на V&V модела

- Дейности за реализация на V&V модела в другите фази на жизнения цикъл
  - ✓ Валидация на новите изисквания за промяна за консистентност със съществуващите изисквания.
  - ✓ Валидация на изисквания за пълното при премахване на съществуваща функционалност.
  - ✓ Верификация на реализацията на новите подсистеми и/или компоненти.
  - ✓ Валидацията на интеграцията.
  - ✓ Крайна валидация на новата версия от реализация на изискванията на възложителя.

Методология за V&V



Как да разбираме препоръките по темата в литературата

- Has To / Must
  - ✓ Това е най-високото ниво задължителност за дадената препоръка, защото описаният процес, процедура или подход *РАБОТИ САМО ПО ТОЗИ НАЧИН*.
- Shall
  - ✓ В този случай има няколко възможни потенциални варианта, но препоръчаният е най-желателен. Много често препоръката се дължи и на данни, които не са част от описанието.
- Should
  - ✓ Това е случая, когато авторът е експерт в дадената област и препоръката отразява опита му, че предложеното решение е най-доброто и най-правилното.

Инженерна дефиниция за система

- Verification, Validation and Testing of Engineered Systems, Wiley
  - ISBN 047052751X
- Guide to software verification and validation
- Verification and Validation in Systems Engineering - Assessing UML-SysML Design Models.pdf
  - Глава 5

Questions ?

Методи за верификация на програмно осигуряване

Артефакти при разработка на ПО

- Технически
  - ✓ Основни – описание на изискванията, на различните проекти (идеен и детайлен), кода на програмата, ръководства за потребителя, за инсталиране
  - ✓ Помощни – модели на обкръжението, формални модели на поведение, тестови набори, т.е. елементи, необходими за провеждане на верификацията.
- Организационни
  - ✓ Различните видове планове – управление на риска, на качеството, график на дейностите и т.н.
  - ✓ Описание на процесите и процедурите за извършване на отделните дейности.

Същност на верификацията

- Оценяват се всички видове артефакти, създавани при разработката и съпровождането на ПО.
- Според видовете артефакти се дели на следните основни групи:
  - ✓ Верификация на изискванията
  - ✓ Верификацията на проектни решения
  - ✓ Верификацията на програмния код
  - ✓ Верификацията на тестовите планове
  - ✓ Верификация на организационни документи и процеси
  - ✓ Верификацията на цялостното разработвано ПО

Задачи на верификацията: изисквания

- Верификация на изискванията – проверява се за няколко различни характеристики на документа за изисквания.
- Най-често според стандартите IEEE 830 и IEEE 1233 :
  - ✓ Еднозначност
  - ✓ Съгласуваност и Непротиворечивост
  - ✓ Пълнота и Минималност
  - ✓ Проверяемост и Проследимост
  - ✓ Систематичност

Задачи на верификацията: проектни решения

- Верификацията на проектните решения оценява:
  - ✓ Връзката “функционалност-изискване”.
  - ✓ Пълнота на документите за разработка.
  - ✓ Коректност и непротиворечивост на документите за разработка.
  - ✓ Коректност на решенията
    - Особено важно за свързаните с минимално задължителните изисквания към разработката, напр. сигурност, надеждност, отказоустойчивост и други.

Задачи на верификацията: програмен код

- Верификацията на програмния код оценява:
  - ✓ Връзката “код-функционалност-изискване”
  - ✓ Спазването на семантичните и синтактични изисквания, т.нар. правила за кодиране (coding rules)
  - ✓ Проверка за различни видове неточности при реализация – цикли, пътища, инициализации и т.н.

Задачи на верификацията: програмен код

- Изследвани типове дефекти:
  - ✓ Използване на данни (data references)
  - ✓ Декларативни грешки (data declaration)
  - ✓ Изчислителни грешки
  - ✓ Грешки при сравнения
  - ✓ Входно/Изходни грешки
  - ✓ Грешки при управление на извършване на действията (control flow)
  - ✓ Интерфейсни грешки
  - ✓ Други

Задачи на верификацията: организационни документи и процеси

- Верификация на организационни документи и процеси
  - ✓ Те се оценяват спрямо задачата на проекта, ограниченията по пари и време.
  - ✓ Оценява се квалификацията на персонала и стратегиите за преквалификация.
  - ✓ Оценява се организационните решения, влияещи върху процеса на разработка.

Задачи на верификацията: цялостно ПО

- Верификацията на цялостното разработвано ПО оценява:
  - ✓ Системата и нейните компоненти могат ли да работят в съответното обкръжение. Описано в заданието
  - ✓ На основата на сценарии се проверява коректността и пълнотата на поведението на системата и компонентите и.

Задачи на верификацията: тестовите планове

- Верификацията на тестовите планове оценява:
  - ✓ Пълнотата на плановете според оценката на риска за проекта
  - ✓ Избраните методи за определяне на дефектите според очакванията за типовете дефекти.
  - ✓ Използваните вторични технически артефакти

Класификация

- Съществуват 4 основни категории:
  - ✓ Неформални (informal)
    - Разчитат изцяло на човешката интерпретация на артефактите и не използват математически формализми. Затова ги наричат и ‘*експертизи*’.
  - ✓ Статични
    - Оценяват статичния дизайн и реализацията му, без оценка на изпълнението върху конкретна машина
  - ✓ Динамични
    - Проверяват поведението на ПО чрез изпълнение върху конкретен компютър.
  - ✓ Формални
    - Използват се различни математически формализми за определяне на дефектите в ПО.

Класификация

Експертизи

- Възникнали са като необходимост от подготовка на специалистите по тестване и настройка (debugging) на ПО
  - ✓ Наричат ги ‘Човешко тестване’ (human testing)
- Изключително ефективни при намирането и отстраняването на дефекти в ПО
  - ✓ Приема се, че във всеки проект трябва да присъстват поне един метод от тази група
  - ✓ Прилага се към всички групи артефакти и за всички етапи
- Имат две основни предимства
  - ✓ Ранното отстраняване на грешките намалява времето за разработка и стойността на проекта
  - ✓ Има чисто психологическо значение за програмистите, водещо до по-малко грешки спрямо работа само с тестване.

Експертизи

- Видове
  - Техническа експертиза (technical rewievs)
  - Инспекции на кода (inspections)
  - Прегледи (walktroughs)
  - Одит (audit)
  - Други
    - Оценка на архитектурата чрез сценарии (scenario based evaluations)
    - Самоконтрол чрез четене (desk-checking)
    - Разглеждане между равни (peer reviews)

Статичен анализ

- Същност:
  - ✓ На основата на формални методи се определя коректността на създаване на различни артефакти, които са част от проекта.
  - ✓ На основата на шаблони, чрез формални техники, се проверява за специфични групи дефекти.
- Основна дилема: “Строгост на анализа – Количество на дефектите”

Статичен анализ

- Предимства:
  - ✓ Осигурява добра възможност за автоматизация на процеса за анализ.
  - ✓ Широко разпространение, особено за някои видове ПО и използвани езици за програмиране и методи за разработка.
- Продукти
  - ✓ PolySpace Verifier, Coverity Prevent и Klocwork K7

Формални методи

- Използват се за анализ на формалния модел на изискванията, поведението на ПО и на обкръжението му.
- Може да се прилага само към артефакти или части от тях, които са изразени формално или за които може да се създаде формален модел.
- Формалните модели не могат да се създават автоматизирано, но техните характеристики могат да се проверяват автоматизирано.
- Много добри за откриването на сложни дефекти, които ме могат да се открият с тестване или експертиза.

Формални методи

- Биват:
  - ✓ Логико-алгебрични модели (property-based models)
    - съждения и тяхното изчисление, предикатна логика и изчисление на изрази в нея (включително и от по-висок ред), λ-изчисления, модални логики, времеви логики, релационна алгебра, алгебрични модели на основата на абстрактни типове, т.н.
  - ✓ Изпълними модели (executable models)
    - виртуални машини, крайни автомати, системи на основата на маркирани преходи, разширени крайни автомати, йерархични автомати, взаимодействащи си автомати, времеви автомати, мрежи на Петри, машини на основата на абстрактни състояния
  - ✓ Междинни модели
    - Логика на Хоар (Hoare), динамични и програмни логики, програмни контракти

Динамични методи

- Изисква наличието на работещо ПО (може и отделни части от него).
  - ✓ Може да се използват и прототипи - в този случай се говори за ‘имитационна динамична верификация’.
- Могат да се намерят дефекти, свързани само с работата на програмното осигуряване, но не и с други етапи от разработката, напр. съпровождане, инсталация и т.н.
- Изискват допълнителна подготовка
  - ✓ Най-често това са тестови набори, симулационна среда, мониторнова система, система за записване на резултатите.

Динамични методи

- Особено подходящ подход за определяне на някои от характеристиките на ПО
  - ✓ Функционалност
    - При проверка на функционалността се следят получаваните резултати, вътрешни състояния, обменяни съобщения, ред на изпълнение на дейностите, по-рядко и времеви зависимости.
  - ✓ Преносимост
    - При проверка на преносимостта се следи за тези елементи на функционалните изследвания, които се влияят от средата на работа на ПО.

Динамични методи

- Особено подходящ подход за определяне на някои от характеристиките на ПО
  - ✓ Производителност
    - Записват се и се оценяват времето за изпълнение на операциите, използваната памет (всички видове), натоварването на комуникационните канали, вътрекомпонентните обмени на информация
  - ✓ Надеждност
    - Записва се броя и времето на поява на дефектите (по класове, по ресурси, по компоненти, по подсистеми).
    - Изпозлват се сценарии за симулиране на промени в обкръжаващата среда (напр. отказ на оборудване).

Динамични методи

- Биват:
  - ✓ Мониторинг
    - Това са дейности по следене работата и записване на текущото състояние на системата и получаваните резултати.
    - След завършване на записването се провежда оценка на получените резултати от мониторинга като средство за определяне на неправилно функциониране.
  - ✓ Тестване
    - Мониторинг, но за специално изградени сценарии на използване.
    - Генерирането на сценариите е с цел създаване на ситуации, при които ПО може да има неправилно поведение.

Синтетични методи

- Основната идея е съчетаване на предимствата от основните видове за сметка на ограничаване на недостатъците им.
- Най-често това е съчетание на динамични с формални методи:
  - ✓ Тестване на основата на модел на системата (model-based testing, model driven testing)
  - ✓ Мониторинг на формални свойства (runtime verification, passive testing)

Инспекции и Прегледи

- Извършва се задължително от група от специалисти, които четат или визуално проверяват програмата (според оценяваните артефакти).
- Участниците имат ясно дефинирани задачи, които трябва да извършат в процеса на работа.
- Основната фаза е ‘Срещата на участниците’
  - ✓ Обсъждат се резултатите от индивидуалната работа като начин за намиране на дефектите и проблемните зони.
  - ✓ Участниците нямат за задача да набележат начините за премахване на грешките.

Инспекции и Прегледи

- Предимства:
  - ✓ Много по-ефективни от самоконтрола, защото се извършва от външни хора, а не от автора.
  - ✓ Намаляват разходите за настройка, защото локализират мястото на дефекта много точно.
  - ✓ Добри са за определяне на логически грешките при разработката или грешки в кода.
  - ✓ Особено полезни при промени на съществуващо ПО.
- Недостатъци
  - ✓ Не са добри за определяне на грешките от етапа на събиране и анализ на изискванията, т.е. в дизайна от високо ниво.

Инспекции

- Дефиниция
  - Това са процедури и техники за откриване на дефекти в ПО на основата на групово изследване чрез четене.
- Същност:
  - ✓ Критическо изследване на програмните артефакти, целящо откриването и отстраняването на дефектите.
- Една от най-използваните техники за верификация и подобряване на качеството на ПО.
- Директно определя и коригира дефектите в ПО.
- Могат да се прилагат за много видове програмни артефакти.

Инспекции

- Същност:
  - ✓ Критическо изследване на програмните артефакти, целящо откриването и отстраняването на дефектите.
- Една от най-използваните техники за верификация и подобряване на качеството на ПО.
- Директно определя и коригира дефектите в ПО.
- Могат да се прилагат за много видове програмни артефакти.

Инспекции: особености

- Реализира се от група *инспектори* в рамките на паралелни и координирани проверки.
- Реализира се на няколко последователни фази, между които има координационни сесии.
- Дефекти се откриват като резултат от индивидуална инспекция, обобщаване на резултатите в сесиите или паралелна инспекция на една и съща част от ПО.
- Дефектите се отстраняват от разработчика и се извършва повторна проверка на ПО.
- Дълбочината на инспекция може да варира според нуждите и типа на ПО.

Фаганови Инспекции (Fagan inspections)

- Една от най-ранните системи е разработена от Fagan през 1976 г.
  - ✓ Планиране – какво ще се инспектира, кой ще участва и каква ще му е ролята.
  - ✓ Начална среща – разпределят се инспекторите по задачи и се определя как ще се извършва работата
  - ✓ Реализация – всеки инспектор подготвя списък с потенциални дефекти и проблемни области
  - ✓ Инспекционно събиране – формира се списъка с дефектите чрез обобщаване и обсъждане на индивидуалните инспекции.
  - ✓ Фиксиране на дефектите - разработчикът отстранява намерените дефекти.
  - ✓ Контролна проверка – прави се валидация на ПО след фиксирането на дефектите.

Инспекции

- Промени в метода на Фаган
  - ✓ Различни техники за реализация на индивидуалните инспекции с цел намаляване на фалшивите съобщения и интензифициране на работата.
  - ✓ Разлики в големината на групите, задачата на модератора и на сесийния ръководител.
  - ✓ Промени в техниките за определяне дефектите – прилагане на все повече систематични техники и намаляване на ad-hoc дейностите.
  - ✓ Използване на резултатите на инспекциите като основа на добри практики за разработка на ПО

Инспекции: фази на процеса

- Планиране и реализация
  - ✓ Каква са целта и обекта на инспекция ?
  - ✓ Какво ще се инспектира ?
  - ✓ Кой ще извършва инспекцията ?
  - ✓ Кой трябва допълнително да се привлече, в каква роля и с какви отговорности ?
  - ✓ Какъв ще е процесът, какви техники ще се използват и какъв ще е контролът на резултатите ?
- Инспекция и определяне на дефектите
- Корекция и контрол на резултатите.

Инспекции: състав

- Модератор
  - ✓ Висококвалифициран програмист, но не разработчика на програмата.
- Програмист
  - ✓ Разработчикът на инспектираното ПО
- Системен дизайнер
- Специалист по тестването

Прегледи на ПО (walkthroughs)

- Подобно е на инспекциите, но процедурите и техниките за откриване на дефекти са различни.
- Общи характеристики
  - ✓ Провежда се от група от 3-5 разработчика, като само един е автор на проверяваното ПО.
  - ✓ Задължителните роли са модератор, тестер, програмист и секретар.
  - ✓ Допълнително участват: много висококвалифициран програмист; специалист по програмни езици; програмист извън проекта; този, който ще поддържа програмата; може и други програмисти от проекта.
  - ✓ Сесиите са по 2-3 часа без прекъсване.

Прегледи на ПО (walkthroughs)

- Реализация
  - ✓ Всеки от участниците предварително е получил и проучил съответния материал
  - ✓ По време на срещата се симулира работата на компютъра.
  - ✓ Тестерът предварително е приготвил група тестове, които се проиграват умствено. Така тестовите данни преминават (*walk through*) през логиката на програмата.
  - ✓ Състоянието на данните и системните ресурси се следи непрекъснато на дъска.

Прегледи на ПО (walkthroughs)

- Цел:
  - ✓ Подготвените тестове не трябва да обхващат дефектите, а само да дадат основа за виртуални експерименти с програмата.
  - ✓ Намерените дефекти не се адресират към програмиста, а се записват за по-нататъшен анализ.
- Отстраняване и контрол на дефектите
  - ✓ По подобие на инспекциите и този процес определя с голяма точност мястото, типа и причината на грешката.
  - ✓ Проверката след поправката е аналогична с инспекциите.

Самоконтрол чрез четене (desk-checking)

- Същност:
  - ✓ Авторът на програмата прилага техника от инспекцията или прегледа върху собствения си код.
- Недостатъци
  - ✓ Много по-неефективен
    - × Човек трудно открива собствените си грешки – стилът на мислене е един и същ.
    - × Няма груповия поглед, позволяващ едновременно да се огледа от всички страни един проблем.
    - × Психологически – никой не иска да си изтъква собствените грешки.
  - ✓ Няма дисциплиниращ ефект върху разработчиците.

Разглеждане между равни (peer reviews)

- Същност:
  - ✓ Техника за определяне на критерии за оценка на качеството, полезността, възможността за разрастване и съпровождане, яснотата
- Реализация
  - ✓ Администраторът на процеса подбира група програмисти, като всеки представя по няколко собствени работи.
  - ✓ Работите се разглеждат анонимно и случайно разпределени, като всеки дава оценки по предварително дефинирана таблица.
  - ✓ При събирането всеки защитава и обяснява оценките си и така се създава обща система за оценяване и критерии за прилагането и.

Мониторинг

- Мониторингът е подход от групата на динамичните техники.
  - ✓ monitoring, runtime verification, online verification, passive testing
- Когато се следи само на производителността се нарича ‘*профилиране*’ (profiling).
- Използваните техники зависят от:
  - ✓ От типа и предназначението на записаната информация.
  - ✓ От типа на получаване на информацията за следените данни.
  - ✓ От типа на получаване на оценъчни характеристики.

Мониторинг

- Влияние на типа и предназначението на записаната информация
  - Общи данни и метрики, свързани с проверяваните характеристики
    - параметри на извикване и резултати, обем и количество на динамичното заделяне на памет, количество на съобщенията и др.
  - Данни, които могат да служат за определяне на грешки и проблемни ситуации
    - динамична памет и указатели, синхронизационни механизми, статистика на мрежовите комуникации/сесии

Мониторинг

- Влияние отт типа на получаване на информация за следените данни.
  - ✓ Получаване от изходния код или получаване от двоичния код
  - ✓ Използваните техники биват:
    - Ръчни
    - Компилаторно-асистирани
    - С използване на бинарни транслации
    - Инжекция на времево изпълнение
  - ✓ Симулаторен мониторинг

Мониторинг

- Влияние от типа на получаване на оценъчни характеристики.
  - ✓ Базирани на събития
  - ✓ Базирани на статистики

Въпроси ?

Тестване на ПО

Тестването като задача

- Нуждата от тестване се определя от желанието за повишаване на стойността на продукта, който се продава.
  - ✓ Тестването се използва за подобряване на качеството и надеждността като процедура за откриване на налични грешки в създаваното ПО.
  - ✓ Повишаването на стойността е за сметка на подобряване на качеството и надеждността.
- Тестването е техническа задача, но тя включва важни аспекти, свързани с икономика и човешка психология.

Психология и Тестването

- Неправилно разбиране на предназначението на процеса на тестване на разработваното ПО.
  - ✓ “Тестването е процес на показване отсъствието на грешки.”
  - ✓ “Целта на тестването е да покаже, че програмата изпълнява необходимите си функции коректно.”
  - ✓ “Тестването е процес на създаването на убеденост, че програмата прави това, за което е предназначена.”

Тестването е процес на изпълняване на програма с цел доказване на грешки в нея на основата на създадени сценарии.

Психология и Тестването

- “Тестването е процес на показване отсъствието на грешки.”
  - ✓ Понятията ‘Успешен тест’ или ‘Неуспешен тест’
- “Целта на тестването е да покаже, че програмата изпълнява необходимите си функции коректно.”
  - ✓ Невъзможността за реализиране на задачата определя задълбочеността на реализирания процес.
- “Тестването е процес на създаването на убеденост, че програмата прави това, за което е предназначена.”
  - ✓ Понятието ‘не работи правилно’
    - ✗ Не само това, което не прави по задание, а и това, което прави повече отколкото трябва.

Психология и Тестването

- Познаването на идеологията на създаване на тестваната програма
  - ✓ Тестването трябва да се извършва от човек, който не е писал програмата, не е запознат детайлно с програмата и не е част от групата, която е създавала програмата.
- Дисциплиниращо действие
  - ✓ Никой не иска да съществува документ, в който са описани неговите грешки.

Икономически аспекти на тестването

- Не може да се определят всички грешки.
  - ✓ Практически е невъзможно да се генерират всички възможни сценарии даже и на малки програми.
- Тестването е времеемък и скъп процес.
  - ✓ Няма физическо време за изпълнение на всички сценарии.
  - ✓ В много случаи стойността на симулация на среда, в която да се прояви някоя грешка може да е много висока, т.е. икономически неизгодно да се прави.
- Някои грешки са по-важни, други са по-маловажни.
  - ✓ Това е пряко свързано с основните характеристики и задачи на разработката.

10-те принципа на тестването

- #1: Дефинирането на очаквания резултат/изходни данни са задължителна част от тестова процедура.
- #2: Програмистите трябва да избягват желанието да тестват собствените си програми.
- #3: Групата, създала ПО не трябва никога да отговаря за тестването на собствените си програми.
- #4: Резултатите от всеки тест трябва да се изследват внимателно и задълбочено.
- #5: Генерираните тестове трябва да обхващат както случаите с невалиден или неочакван входен набор от данни, така и с валиден и очакван входен набор.

10-те принципа на тестването

- #6: Проверката дали програмата прави това, за което е създадена е половината от работата. Другата половина е за проверка дали програмата прави неща, за които не е предназначена.
- #7: Избягвайте безсмислените тестове освен ако програмата не е безсмислена.
- #8: Не планирайте тестове с презумпцията, че няма грешки.
- #9: Вероятността за наличие на още грешки в дадена част от програмата е пропорционална на броя на намерените вече грешки.
- #10: Тестването е изключително интелектуален и творчески процес.

Генерация на тестовия набор

- Независимо от големината и спецификата на генерирания тестов набор не може да се гарантира, че с него ще се открият всички грешки.
- Ограничението по време и стойност на процеса е наложило правилото:
  - ✓ От всички възможни входни набори трябва да се определи такова подмножество, което да позволи установяването на най-много грешки.
- Съществуват няколко различни стратегии:
  - ✓ Случаен подбор на подмножеството от тестови данни.
  - ✓ Метод на черната кутия
  - ✓ Метод на бялата кутия
  - ✓ Комбиниран метод

Видове

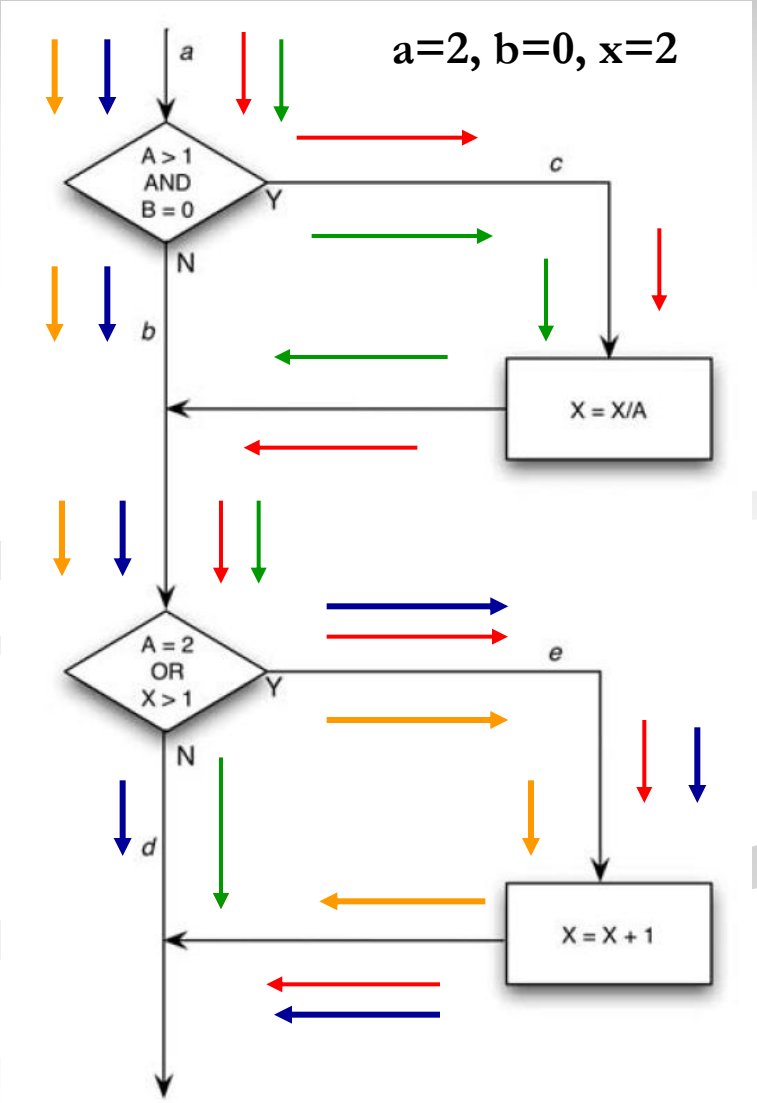
- Метод на бялата кутия
  - ✓ Основан на покритие на логиката на програмата
    - × Покриване на оператор (statement coverage)
    - × Покриване на решение (decision coverage)
    - × Покриване на условие (condition coverage)
    - × Покриване ‘решение -условие’ (decision-condition coverage)
    - × Покриване ‘множествено условие’ (multiple-condition coverage)
- Метод на черната кутия
  - ✓ Еквивалентно (равнозначното) разделяне
  - ✓ Анализ на гранични стойности
  - ✓ Графи ‘Причина-Ефект’

Логическо покриване

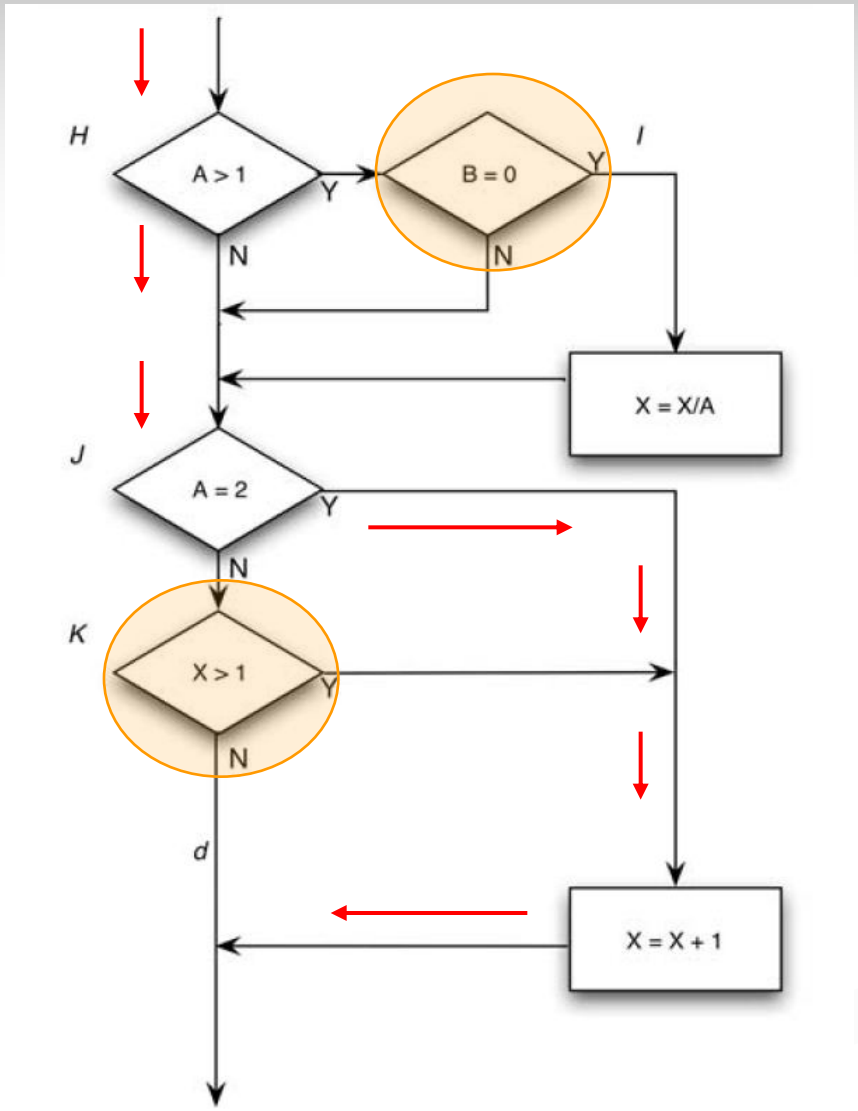
```
void foo (int a, int b, int x) {
    if (a>1 && b==0) {
        x=x/a;
    }
    if (a==2 || x>1) {
        x=x+1;
    }
}
```

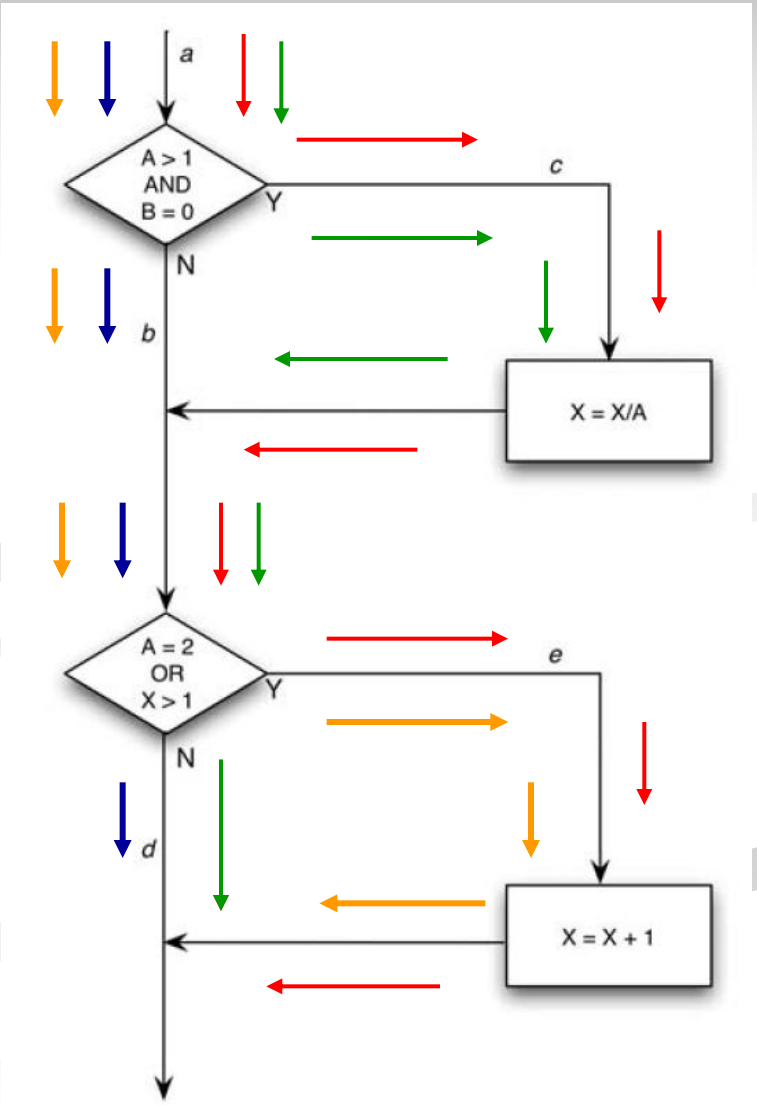
Логическо покриване

```
void foo (int a, int b, int x) {
    if (a>1 && b==0) {
        x=x/a;
    }
    if (a==2 || x>1) {
        x=x+1;
    }
}
```



Логическо покриване





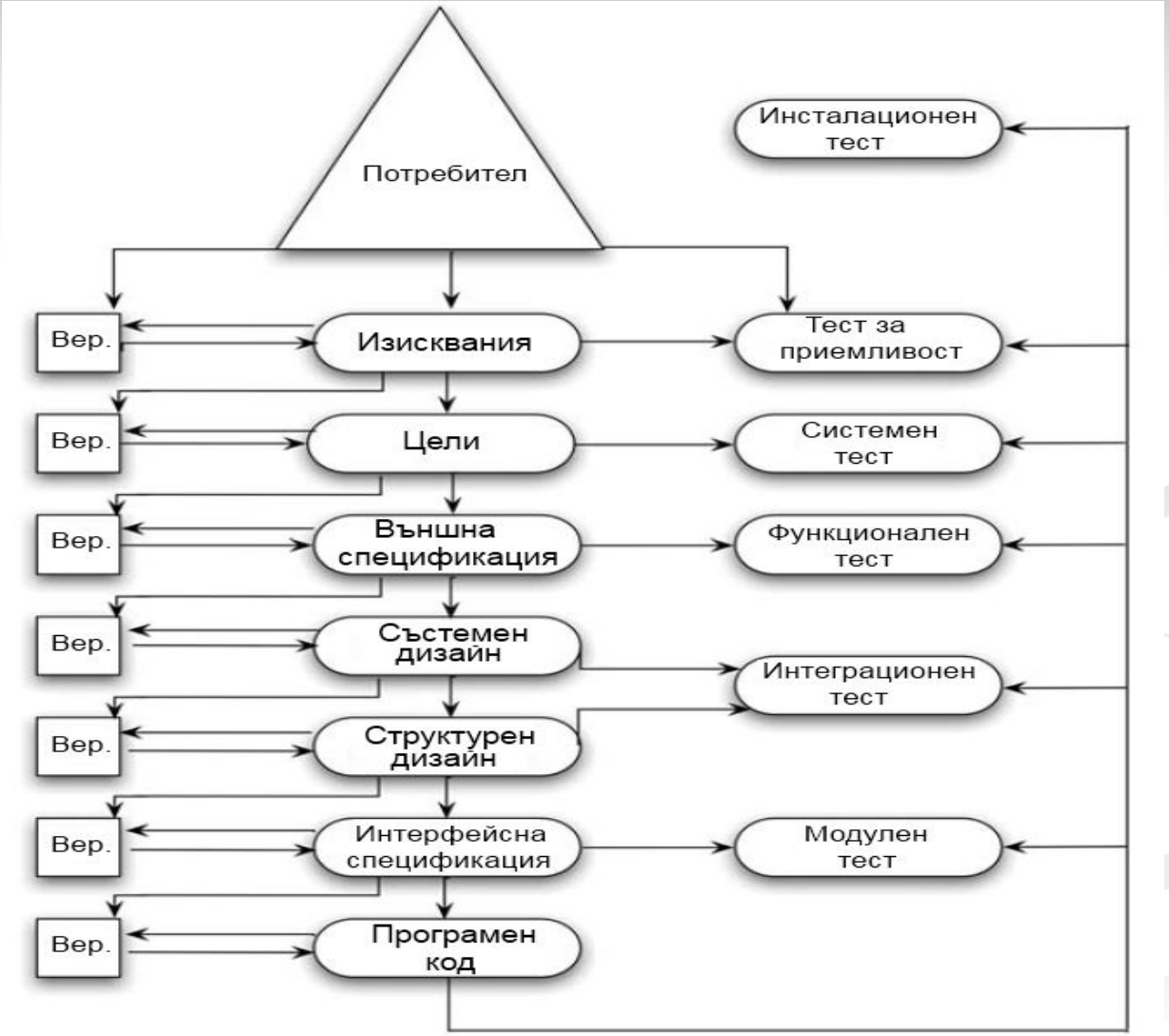
Стратегии при избора на тестов набор

1. Ако спецификацията съдържа комбинация от входни условия най-добре е използването на графи ‘Причина-Ефект’
2. Винаги да се използва анализа на граничните стойности – независимо че тези стойности могат да се получат и от графите ‘Причина-Ефект’.
3. Идентифицирайте валидните и невалидните равнозначни класове за входните и изходните данни, и ако е необходимо да се допълни тестовия набор, създаден от предходните стъпи.

Стратегии при избора на тестов набор

4. На основата на ‘отгатване’ да се попълни тестовия набор с още тестове.
5. Анализирайки логиката на програмата се генерират допълнителни тестове за случаите на комбинация от условия, които не изглеждат невъзможни и които не са били създадени на предните етапи (използват се всички обхват).

Нива на тестване на ПО

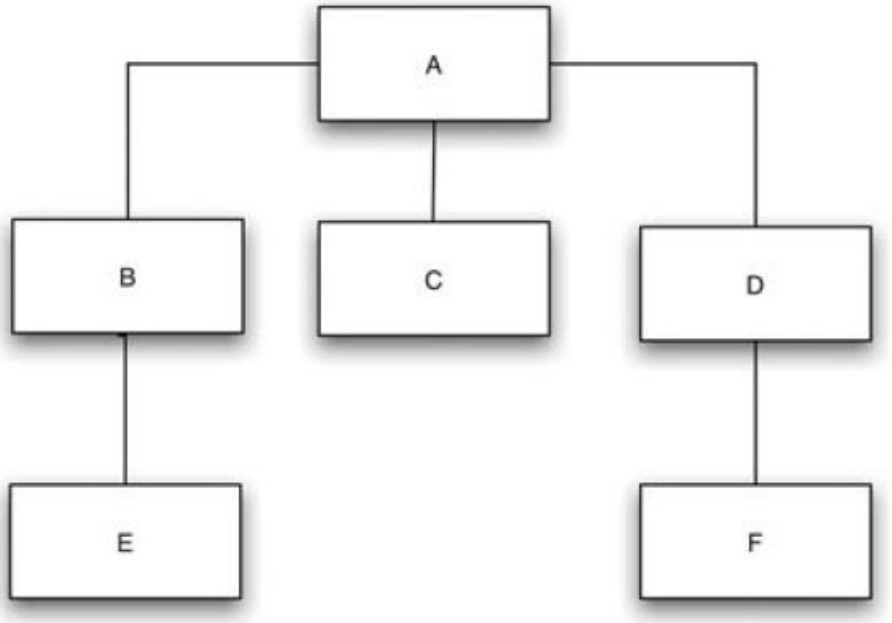


Тестване на модули

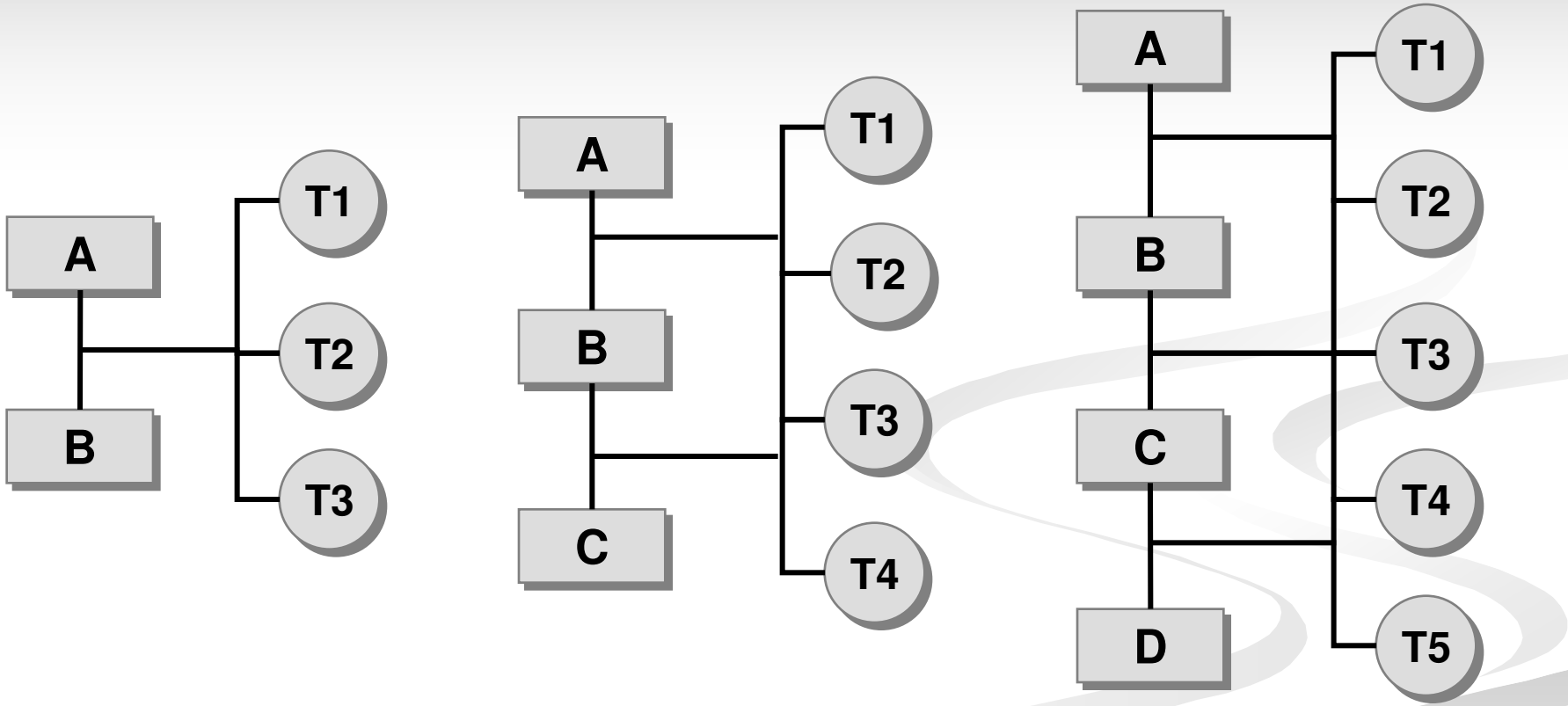
- Тестването на модули (unit testing) е насочено към проверка на отделните блокове (подсистеми, функции и др.) самостоятелно.
- Основно е ориентирано към проверка по метода на бялата кутия.
- Предимства:
  - ✓ Лесна комбинация на методите – елементите са малки по размер и сложност.
  - ✓ Паралелност на обработката на отделните елементи.
  - ✓ Лесна локализация на грешката при използване на резултатите от тестването.

Инкрементално тестване

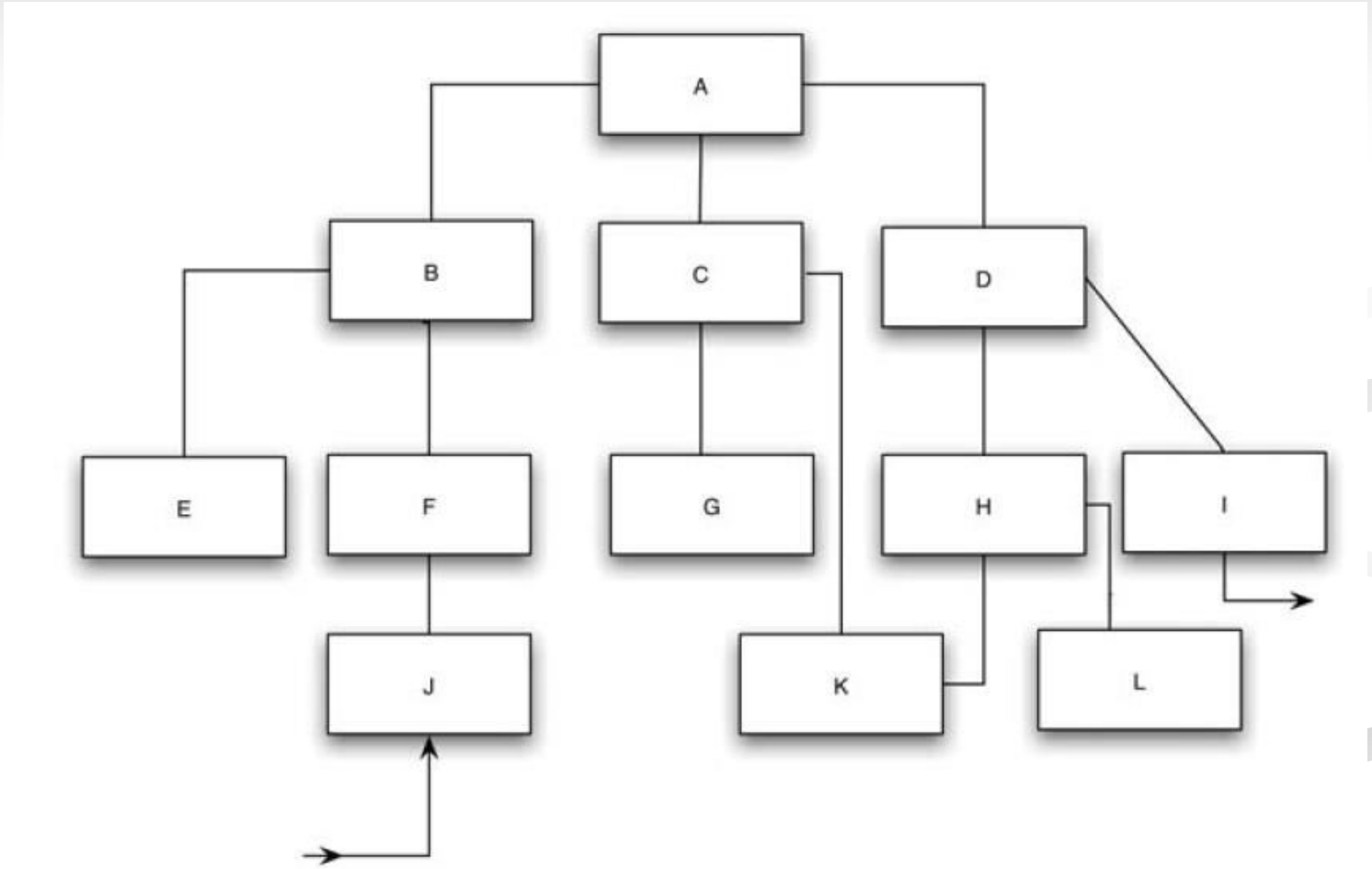
- Това е техника за тестване на модули, при която дадения модул се обвързва със съседен/-и модул/-и (вече тестван/-и) и се тестват заедно.
- Два подхода
  - ✓ Отдолу-нагоре
  - ✓ Отгоре-надолу



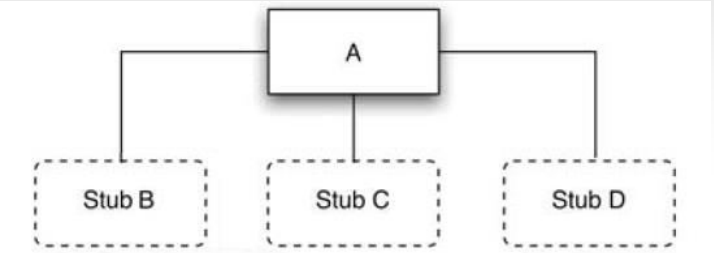
Инкрементално тестване

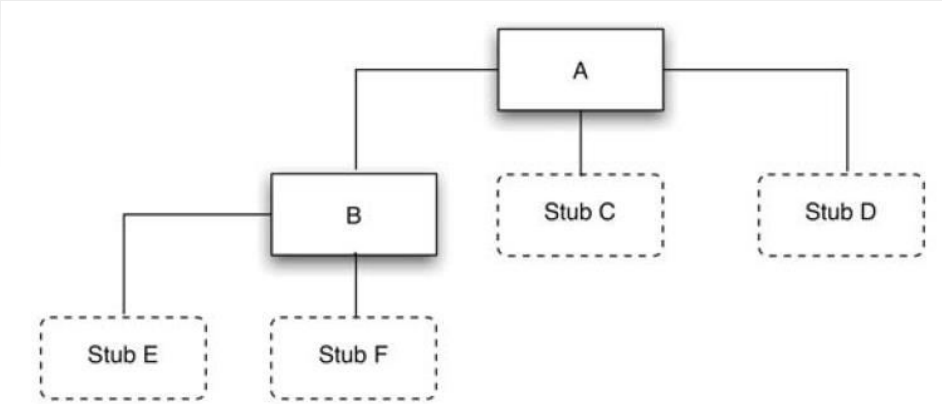


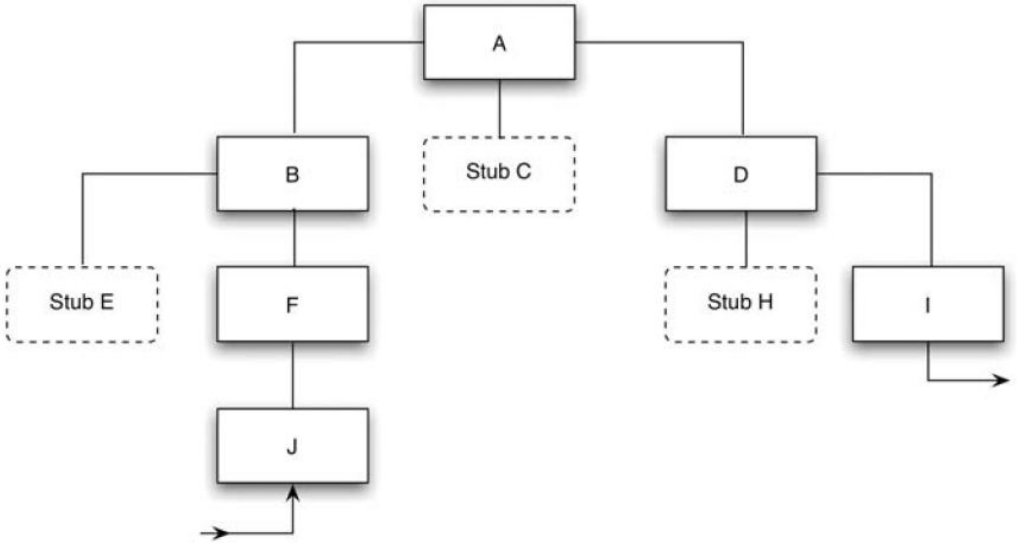
Инкрементално тестване



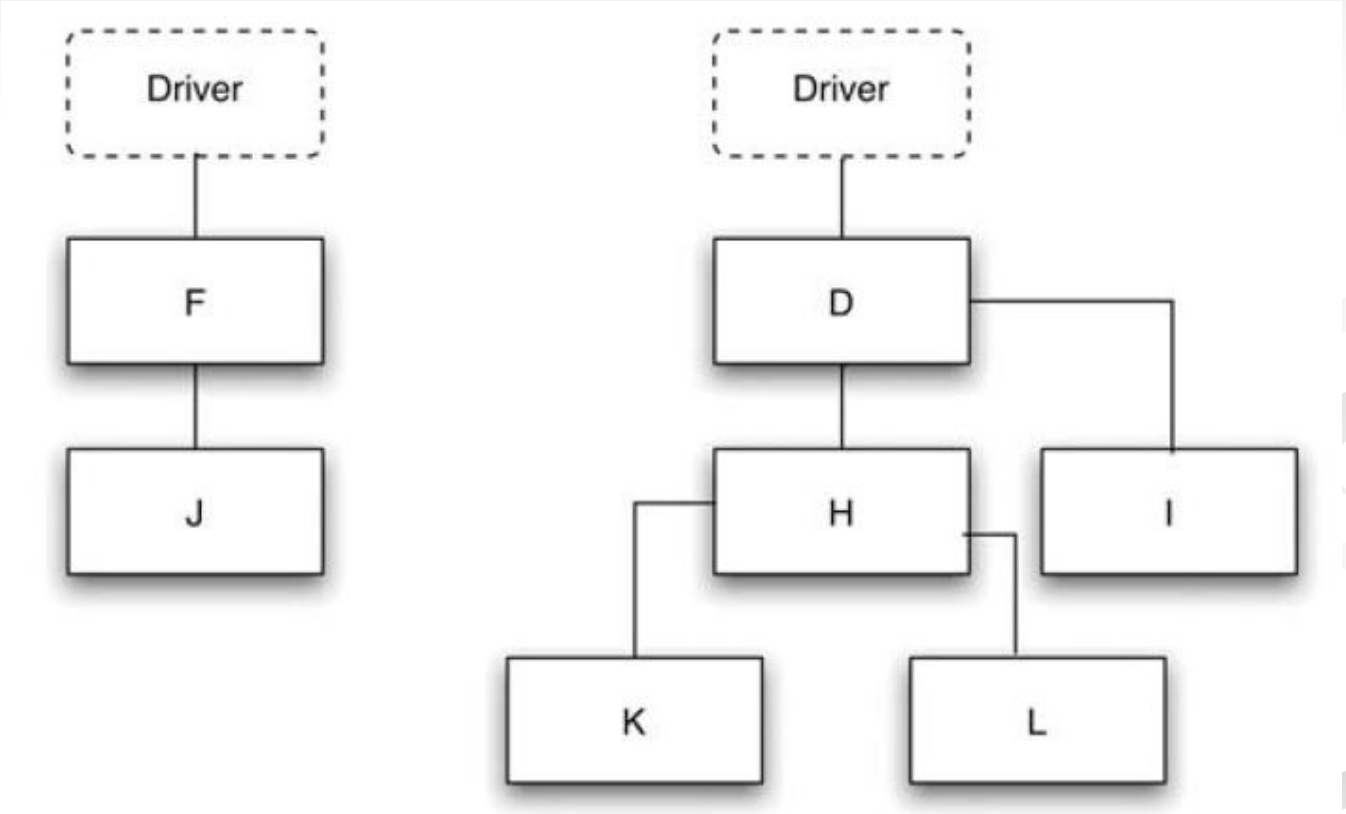
Отгоре-надолу







Отдолу-нагоре



Тестване на ПО на по-високо абстрактно ниво

- Модулното тестване е първия етап от процеса.
  - ✓ Това е особено вярно при големи и/или слоцни системи.
- Тестваното на по-високо абстрактно ниво се дължи на факта, че задачата на ПО е да трансферира информация между елементите и/или да преобразува тази информация от един вид в друг.
  - ✓ Грешките най-често са резултат от прекъсване на предаването на информацията, неправилно преобразуване или появата на шум.

Функционален тест

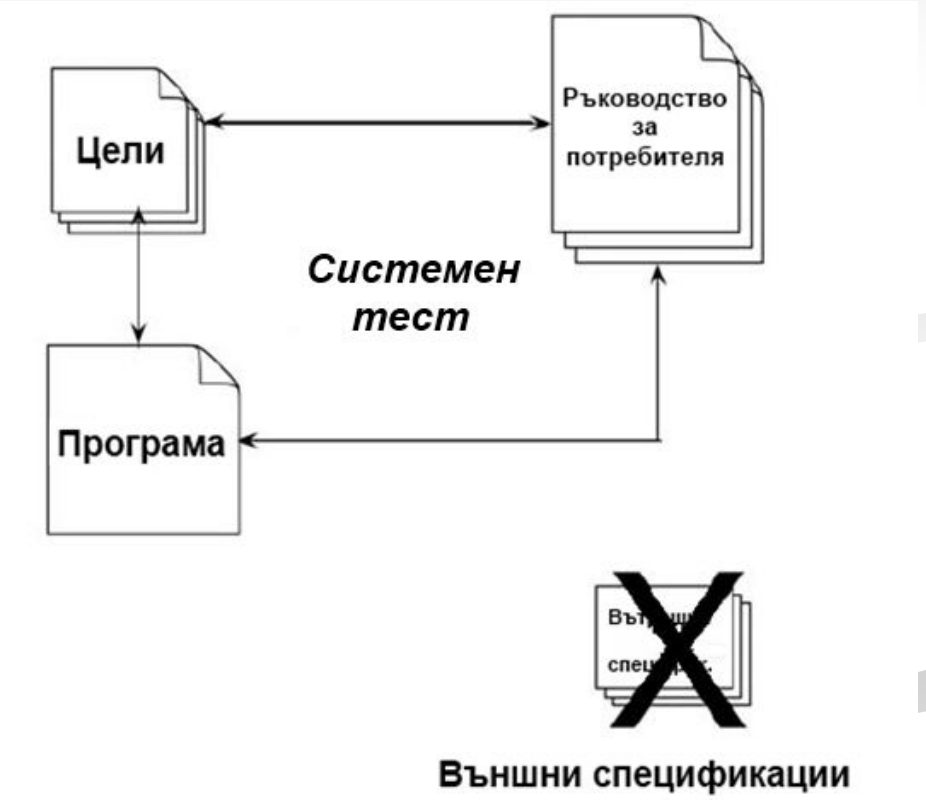
- Предназначение:
  - ✓ Откриване на различия между програмата и външните спецификации, задаващи поведението на програмата от гледна точка на потребителя.
- Реализация
  - ✓ При малки програми се използва метода на черната кутия.
  - ✓ При по-големи програми се използват методи от групата на бялата кутия (по същество всички без логическо покритие).
- Препоръки
  - ✓ Да се доразвива тестването по посока на сценариите с най-много открити грешки.
  - ✓ Специално внимание да се обърне на невалидни и/или неочаквани входни данни.

Системен тест

- Предназначение
  - Задачата му не е да проверява функционалността на системата като цяло – това е задача на функционалния тест.
  - Системният тест е насочен да провери как системата (като цяло) реализира определените и цели.
    - Задължително трябват метрики за измерване на степента на реализация на целите.
    - Задължително се извършва като част от дефинираната работна среда.
- Извършва се от външна група с опит както като разработчици, така и като потребители.

Системен тест

- Генерация на тестовите набори



Системен тест

- Видове:
  - ✓ Реализирани обработки (facility testing)
  - ✓ Тестване по размер (volume testing)
  - ✓ Тестване с екстремно натоварване (stress testing)
  - ✓ Тест за използваемост (usability testing)
  - ✓ Тест за сигурност (security testing)
  - ✓ Тест за производителност (performance testing)
  - ✓ Тест за съхраняемост на информацията (storage testing)
  - ✓ Конфигурационен тест (configuration testing)

Системен тест

- Видове:
  - ✓ Тест за съвместимост (compatibility/configuration/conversion testing)
  - ✓ Тестване на възможност за инсталиране (installability testing)
  - ✓ Test за издръжливост (reliability testing)
  - ✓ Тест за възстановяемост (recovery testing)
  - ✓ Тест за системна поддръжка (serviceability testing)
  - ✓ Тест на документацията (documentation testing)
  - ✓ Тест за част от процедури (procedural testing)
  - ✓ Конфигурационен тест (configuration testing)

Тест за приемливост

- Тестът за приемливост (acceptance testing) е проверка как програмата реализира наразлите си изисквания.
- Най-често се извършва от крайния потребител, т.е. разработчиците най-често не отговарят за организацията на този процес.
- Стратегия за реализация:
  - ✓ От документа с изискванията се проверяват едно по едно кое не е реализирано.

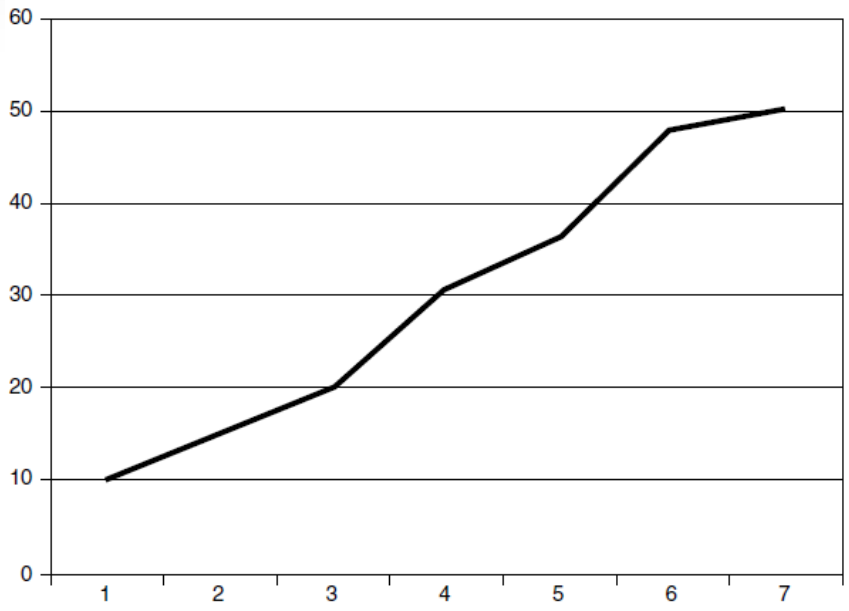
Инсталационен тест

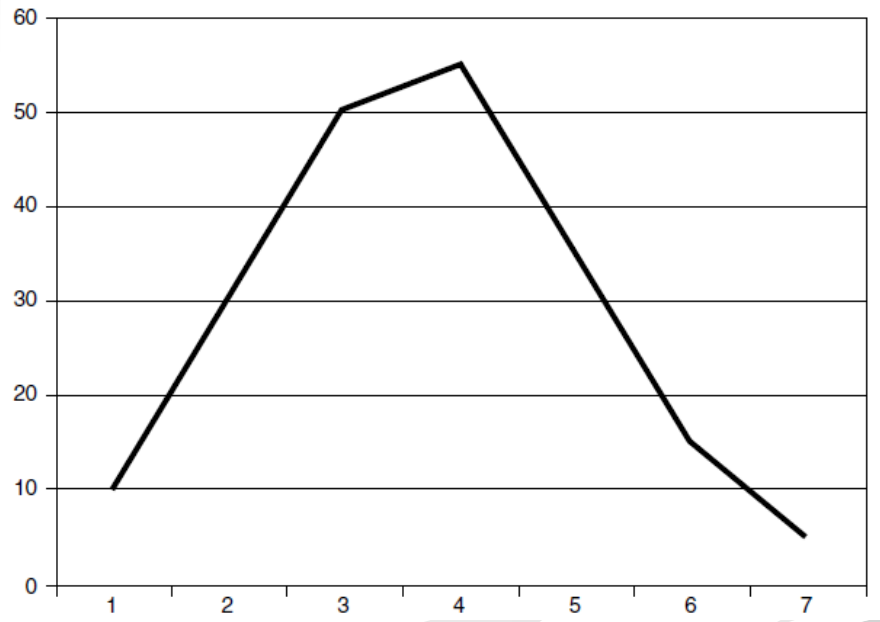
- Инсталационният тест (installation testing) е специален тип тестване, защото при него не се търсят програмни грешки, а се проверява за грешки, резултат от процеса на инсталиране на програмата на конкретна система.
- Основни грешки:
  - ✓ Валидни опции при конфигуриране на системата за инсталация.
  - ✓ Задаване на валидна апаратна среда.
  - ✓ Нужда от комуникация с други програми в процеса на инсталация.
  - ✓ Дали се прехвърлят и/или използват необходимите файлове и библиотеки.
- Създава се и се изпълнява от разработчиците.

Кога свършва тестването

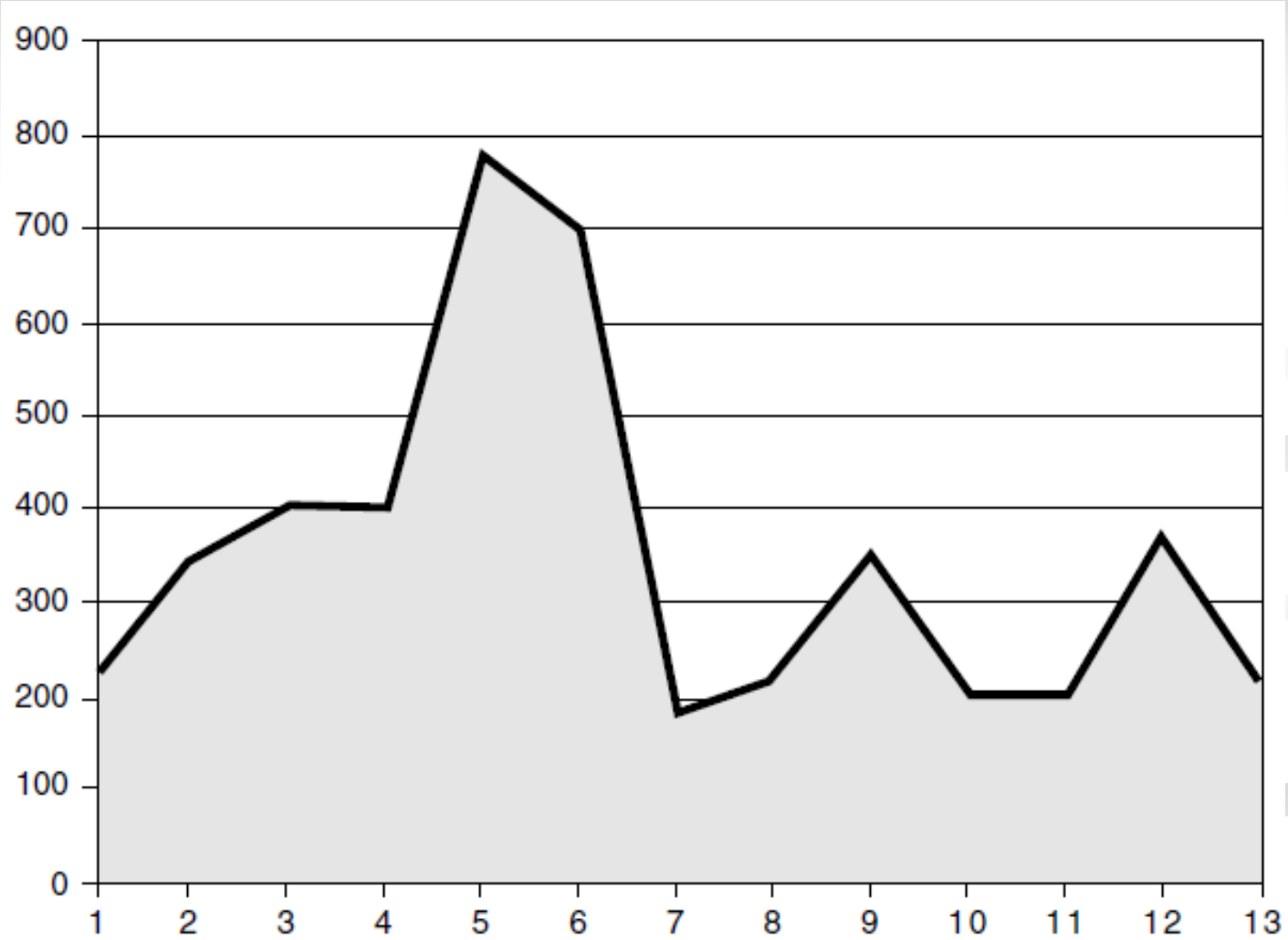
- Това е един от най-важните въпроси
  - ✓ Знае се, че не може да се намерят всички грешки.
  - ✓ Не се знае какъв процент от грешките е намерен.
  - ✓ Има сериозна стойност, т.е. струва пари.
  - ✓ Сериозни проблеми с графиците, като най-често сериозно забавя етапите.

Кога свършва тестването





Кога свършва тестването



Използване на външна група

- Много честа практика, защото някои видове тестване е не трябва да се организират и извършват от разработчиците.
- Изисква се групата да е специализирана за нашия клас ПО, иначе резултатите от тестването ще са приемливи само за определени тестове.
- Необходимо е подготовка на изчерпателна документация преди да се започне процеса.
- Изполва се за доказване на качество на разработваното ПО.

Въпроси ?

Валидация на Програмно Осигуряване

Дефиниция

- Валидацията на програмно осигуряване е задължителен елемент от дейностите и процедурите по подобряване на качеството на създаваното програмно осигуряване.
  - ✓ “Процес на оценяване на система или нейни компоненти в процеса на разработка или в нейния край за определяне дали задоволяват специфицираните изисквания”
  - ✓ “Процес на събиране на информация, показваща че ПО и асоциираните продукти задоволяват системните изисквания в края на всеки цикъл от живота му за решаване на някой сериозен проблем, както и да задоволяват потребителски нужди и зададено използване.”

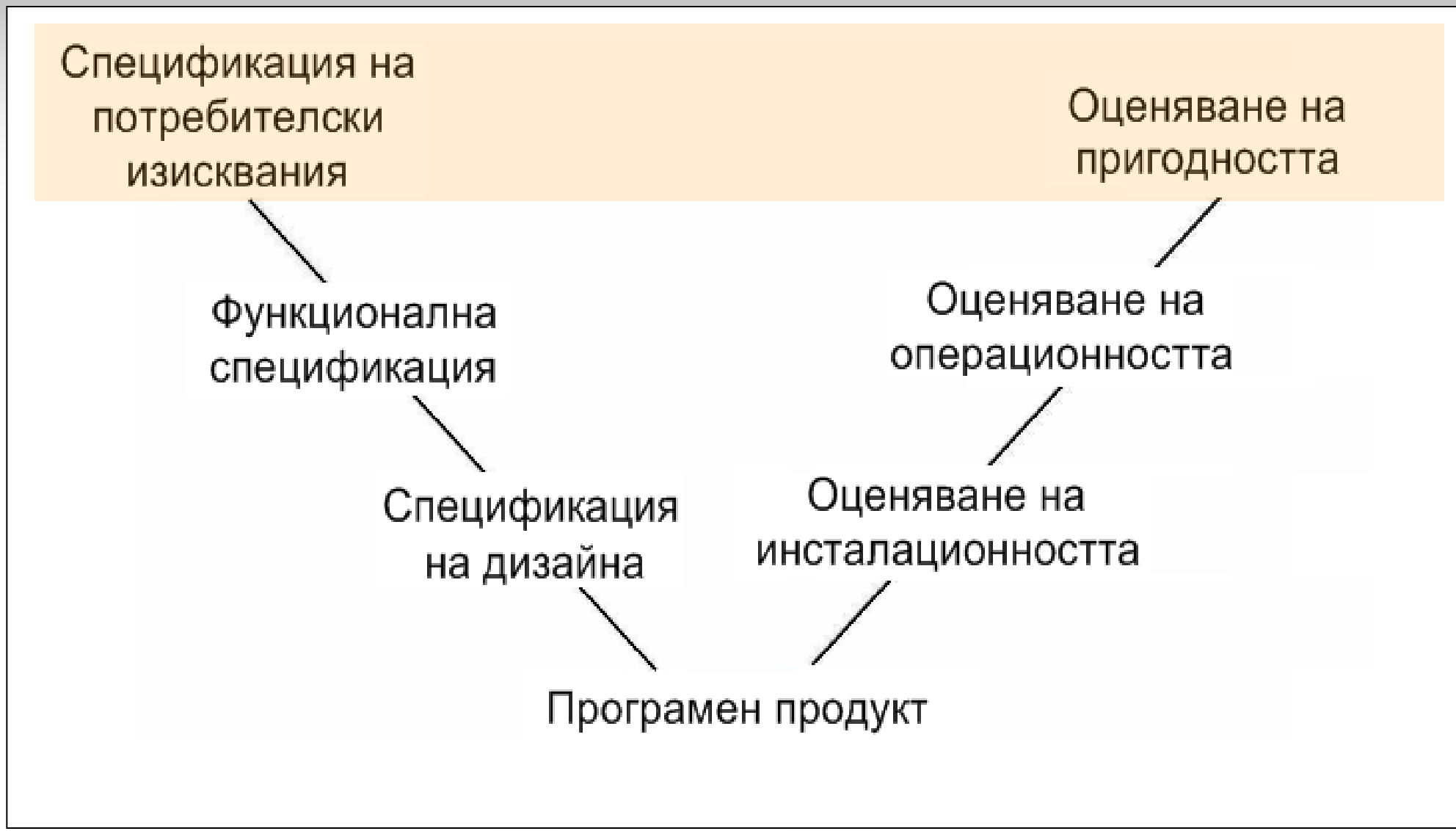
Предимства от използването му

- Подобрява използването на отделните технологии.
- Подобрява бизнес-резултатите.
- Подобрява връзките между ‘Възложител-Производител-Потребител-Засегнати’.
- Подобрява ефективността на използването.
- Намалява риска за откази и грешки.
- Изисква структурен подход към разработката.
- Подобрява обвързаността с регулациите в областта.

Същност

- Валидацията е базирана на методологиите за управление на процеса на производство.
- Свързана е с последователните дейности по пътя на разработката.
- Задължително отчита средата, в която ще функционира програмния продукт.

Същност


Спецификация на потребителски изисквания
Оценяване на пригодността
Функционална спецификация
Оценяване на операционността
Спецификация на дизайна
Оценяване на инсталационността
Програмен продукт

Характеристики

- Показва управление на производството и работата на ПО.
- Осигурява съгласуваност.
- Генерира знания и умения.
- Показва потенциални бъдещи изисквания.
- Изисква структурен подход към разработката.

Това е много повече от тестване за приемливост.

Разлика при валидацията на софтуер и хардуер

- При програмните продукти може да се трасира причината за появата на грешката назад до спецификацията.
- Качеството на програмните продукти много по-малко се влияе от средата за производство.
- Големината на софтуерния продукт не е свързана със сложността му.
- Тестването не позволява гарантирането на правилното функциониране.
- Програмният продукт не е физически обект, т.е. той не се износва и не остарява физически.

Разлика при валидацията на софтуер и хардуер

- Дефектите в програмните продукти са резултат от специфични входни набори (могат да са свързани и с определени моменти във времето)
  - ✓ Появяват им е без предварително предупреждение.
- Скоростта и простотата при разработка на ПО много често е причина за възникване на повече дефекти, отколкото дизайна и условията на производство.
  - ✓ Малки промени могат да доведат до сериозни проблеми на различни места в програмния продукт.

Разлика при валидацията на софтуер и хардуер

- Най-често промените в програмните продукти се правят от хора, които не са участвали в оригиналния проект.
- Софтуерните компоненти не са така добре стандартизирани и взаимозаменяеми, както е при хардуерните компоненти.

Принципи на програмната валидация

- Зависимост от спецификацията на изискванията
- Превенция на дефектите
- Оценка на времето и усилията
- Връзка с жизнения цикъл на ПО
- Планиране
- Избор на процедурите за реализация.
- Дълбочина след промяна на ПО.
- Независимост на оценката от производителя.
- Гъвкавост и отговорност на процедурата.

Планиране на качеството

- Основни дейности
  - ✓ Специфициране на задачите за всяка от дейностите в жзнения цикъл.
  - ✓ Определяне и изброяване на важните фактори за качеството.
  - ✓ Определяне на методите и процедурите за всяка задача.
  - ✓ Определяне на критериите за приемане на всяка от задачите.
  - ✓ Определяне на критериите за дефиниране и документиране на резултатите, позволяващо оценка на съответствието им с изискванията.

Планиране на качеството

- Основни дейности
  - ✓ Определяне на връзките по вход и изход за всяка задача.
  - ✓ Определяне на роли, ресурси и отговорности за всяка задача.
  - ✓ Оценка на рисковете и предположения за появата.
  - ✓ Документиране на потребителските изисквания.

Планиране на качеството

- Типичният план съдържа:
  - ✓ План за управление на риска.
  - ✓ План за управление на конфигурациите.
  - ✓ План за осигуряване на качеството
    - План за валидация и верификация
      - Дейности и критерии за приемане на дейностите
      - График и планиране на ресурсите
      - Описание на изискванията за провеждане
    - Изисквания за формалния анализ на дизайна
    - Описание на изисквания за други технически спецификации.
  - ✓ Начин за описание на проблемите и методите им за решаване.
  - ✓ Други съпътстващи дейности.

Спецификация на изискванията

- Спецификациите на софтуерни продукти включват:
  - ✓ Всички входни данни и сигнали.
  - ✓ Всички резултати от работата на ПО.
  - ✓ Всички функции, които ПО трябва да реализира.
  - ✓ Всички изисквания за изпълнение на функциите, които трябва да се спазват:
    - вид, характер и качество на данните, надеждност, времеви ограничения и други.
  - ✓ Дефиниране на всички вътрешни, външни и потребителски интерфейси за ПО
  - ✓ Описание как потребителя ще взаимодейства с ПО

Спецификация на изискванията

- Спецификациите на софтуерни продукти включват:
  - ✓ Как ще се установяват грешките и как ще се управляват.
  - ✓ Максималното време за отговор.
  - ✓ Описание на средата за функциониране на ПО, ако от това зависи дизайнът му (платформи, ОС, БД и др.)
  - ✓ Всички диапазони, гранични стойности, стойности по подразбиране, специфични стойности, които ПО ще получава.
  - ✓ Описание на всички изисквания, спецификации и характеристики, функциониране на системата за сигурност, отказоустойчивост и самовъзстановяемост.

Спецификация на изискванията

- Основни дейности, свързани с валидацията:
  - ✓ Предварителен анализ на риска.
  - ✓ Анализ на възможностите за трасиране
    - × Програмни изисквания – Системни изисквания
    - × Програмни изисквания – Анализ на риска
  - ✓ Описание на характеристиките на потребителя
  - ✓ Оценяване на изискванията
  - ✓ Анализ на изискванията за потребителски интерфейс
  - ✓ Създаване на план за системно тестване
  - ✓ Създаване на план за валидационно тестване
  - ✓ Описание на двусмислиците в документите за анализ и описание.

Спецификация на изискванията

- Минимална форма на валидация:
  - ✓ Няма вътрешна неконсистентност.
  - ✓ Всички изисквания за изпълнение на функциите са описани ясно, пълно и подробно
  - ✓ Изискванията за надеждност, сигурност и отказоустойчивост са коректни и пълни.
  - ✓ Разпределението на функциите е пълно и коректно.
  - ✓ Изискванията са подходящи за избраното ниво на риск.
  - ✓ Изискванията са описани по начин, позволяващ измерване на резултатите и верифициране на целите.

Дизайн

- Основни дейности, свързани с валидацията:
  - ✓ Дефиниране на критерии за приемливост при специфициране на изискванията
  - ✓ Анализ на риска
  - ✓ Специфициране на стандарти и добри практики за разработка на кодиране.
  - ✓ Изисквания за системна документация и документацията за разработка
  - ✓ Изисквания към апаратната част, която ще се използва.
  - ✓ Определяне на параметрите, които ще се измерват.
  - ✓ Изисквания за модела и типа на данни.

Дизайн

- Основни дейности, свързани с валидацията:
  - ✓ Изисквания за логическата структура и използваните алгоритми.
  - ✓ Дефиниране на класовете грешки, алармите и предупредителните съобщения.
  - ✓ Изисквания за качеството на съпътстващото програмно осигуряване.
  - ✓ Изисквания за модела и технологиите за реализация на различните видове интерфейси (вътрешни, външни, с потребителя)
  - ✓ Метрики, методи и методологии за измерване на сигурността
  - ✓ Други изисквания, метрики и ограничения.

Дизайн

- Минимална количество дейности, свързани с валидация на ПО:
  - ✓ Промени в анализа на риска
  - ✓ Анализа на трасировката:
    - Двупосочна връзка ‘Спецификация на дизайн – Изисквания’
  - ✓ Оценяване на софтуерния дизайн
  - ✓ Анализ на дизайна на интерфейсите
  - ✓ Генерация на дизайна на тестването (от ниво модул нагоре).

Реализация и Кодиране

- Минимална количество дейности, свързани с валидация на ПО:
  - Анализа на трасировката:
    - ‘Код – Спецификация на дизайн’
    - ‘Тест – Код’
    - ‘Тест – Спецификация на дизайн’
  - Анализ на дизайна на интерфейсите в програмния код.
  - Генерация на тестовото множество (от ниво модул нагоре).

Тестване

- Метрики на основата на структурно покритие:
  - ✓ Ниво оператор
  - ✓ Ниво разклонение
  - ✓ Ниво условие
  - ✓ Ниво многоусловни разклонения
  - ✓ Цикли
  - ✓ Пътища
  - ✓ Поток на данните

Тестване

- Метрики на основата на функционално покритие:
  - ✓ Нормално използване
  - ✓ Насочено към резултатите
    - Обхващане на всички възможни резултати
  - ✓ Устойчивост на неочаквани входни въздействия
  - ✓ Насочено към входните данни и сигнали
    - Обхваща различни комбинации на входни данни и сигнали.

Тестване

- Минимална количество дейности, свързани с валидация на ПО:
  - ✓ Планиране на тестовете.
  - ✓ Определяне на тестови набори за структурното покритие
  - ✓ Определяне на тестови набори за функционално покритие
  - ✓ Анализа на трасировката:
    - × ‘Модулен тест – Дизайн от ниско ниво’
    - × “Тест за интеграция – Дизайн от високо ниво’
    - × ‘Системен тест – Изисквания’
  - ✓ Изпълнение на тестовете
  - ✓ Оценка на резултатите и провеждането на тестовете.
  - ✓ Определяне на грешките
  - ✓ Създаване на отчет за тестването.

Съпровождане и Промяна на ПО

- Допълнителни дейности, оказващи влияние върху валидацията на ПО:
  - ✓ Преразглеждане на валидационните планове.
  - ✓ Оценка на аномалиите.
  - ✓ Идентификация на проблемите и трасировка на дейностите по отстраняването им.
  - ✓ Оценка на очакваните/направените промени.
  - ✓ Повторение на дейности от предходни етапи на разработка.
  - ✓ Обновяване на документацията.

Специфика и създаване на ПО

- Валидационната процедура зависи от двата основни аспекта в процеса на създаване на ново програмно осигуряване:
  - ✓ Връзката ‘програма-пазарен продукт’
  - ✓ Стратегия за създаване на нов програмен продукт
- Три вида на програмите като пазарен продукт:
  - ✓ когато ПО е част от система и не може да функционира самостоятелно
  - ✓ когато сами по себе си са продукт
  - ✓ когато се използват за създаване на други продукти

Специфика и създаване на ПО

- Съществуват 4 основни начина за създаване на нов програмен продукт:
  - ✓ *Нов продукт* - изцяло новоразработено програмно осигуряване
  - ✓ *Конфигуриране/Преконфигуриране* - ПО на основата на конфигурация и/или генерация по конфигурация от съществуващи модули
  - ✓ *Модифициране на ПО* - когато се използва подобно и съществуващо ПО, което се променя за постигане на новите изисквания и за което цялата документация е налична
  - ✓ *Използване на готово ПО* - когато се използва съществуващо ПО, отговарящо на спецификациите на заданието (или на части от него), но документацията е няма или е много малко

Дейности

- Реализирани чрез ревюта, инспекции и прегледи на различни документи, част от процеса на производство на ПО:
  - ✓ обхват на процедурата по тестване на продукта
  - ✓ резултати от валидационното тестване
  - ✓ документа със записи на аномалиите, открити в ПО
  - ✓ верификационната документация
  - ✓ документа със записите, описващи проблемите при работа и съпровождане на ПО
  - ✓ записите за трасируемост на промените

Нов продукт

- Валидационен процес
  - ✓ Реализира се на основата на пълната документация за разработваното ПО.
  - ✓ Не зависи от валидацията на системите, които съдържат/използват новото ПО.
  - ✓ Започва се най-често от спецификациите на системните изисквания
    - Особено важно е при ‘критични’ системи (socio-critical, safety-critical).
  - ✓ Документа за началото на процеса за валидация трябва да съдържа минимум следните елементи: системните спецификационни изисквания; валидационен тестов план, тестови процедури, тестов дизайн и тестово множество, спецификация на компютърната система.

Нов продукт

- Валидационен процес
  - ✓ Статичната и динамична симулация на входните сигнали трябва да се реализира така, че да се възпроизведе нормалната среда на работа на ПО.
    - нормални операции, проявления, ограничения
  - ✓ При ‘критични’ системи е необходимо да се създаде представителен тест, който да демонстрира реакциите на всеки входен параметър поотделно, както и на комбинациите от тези параметри.
  - ✓ Специално се изследва верификационната процедура(-и) дали е обхванало всички функции на ПО.
  - ✓ Задължително трябва да се създадат условия за проверка на времеви ограничения, ограничения по ресурси и други.

Нов продукт

- Резултати от валидационния процес
  - Валидатиционен отчет
    - Описва състоянието (успешно/неуспешно), като при неуспешно дава обяснение на отделните причини за дефекти.
    - Не дава препоръки как да се коригират откритите дефекти.
  - Отчет за откритите аномалии
    - Задължително описва всички аномалии, възникнали в процеса на валидация на ПО.
    - За всяка аномалия се описват следните данни: описание и място на проявление; оценка на влиянието; причина за появата и същност на грешката; колко е критичен дефекта за ПО; препоръки за действие.
    - При наличие на открити аномалии не може да завърши процедурата по валидация.

Модифициране

- За новите части важат правилата като за ново ПО.
- В случаите, при които се променя функционалността на съществуващото ПО се извършват всички валидационни действия за дадената подсистема/модул, проведени и описани в документацията.
- В случаите на промяна на базовите спецификации на съществуващото ПО се прилага пълна схема на валидация (като за изцяло нов продукт).
- Промените на конфигурационно ниво на продукта най-често се проверяват с верификация на интеграцията.

Модифициране

- За да може да се използва за нуждите на разработката съществуващото ПО трябва да отговаря не следните изисквания:
  - ✓ Разработката трябва да е била на основата на добри практики и заложените метрики в плана за качество трябва да са съизмерими с необходимите за новото ПО.
  - ✓ Съществува документация за историята на работа и използване на ПО, която да показва надеждност на ПО.
  - ✓ Съществува пълна и прецизна документация за потребителя.

Модифициране

- При отсъствие на програмните кодове:
  - Трябва да се извърши анализ на:
    - Функционални и интерфейсни спецификации
    - Документация за дизайна на системата
    - Тестовите данни
  - Анализът трябва да се съсредоточи специално върху използванията на прекъсвания, рекурсии и заделянето на памет и други системни ресурси.
    - Трябва генерирането на нови тестове, проверяващи коректното функциониране в тези случаи.
    - Необходими са тестове за определяне на коректността на управление на ресурсите и за самопроверка (напр. управлението на стека).
    - Това е особено съществено за критични системи и в този случай е препоръчително провеждане на валидационна процедура като за ново ПО.

Използване на части от ПО

- В този случай не може да се разчита на документите за разработката на старото ПО
  - ✓ верификационните документи от етапите на разработка
  - ✓ документация за програмния код
  - ✓ изискванията за програмния продукт
- Анализира се информацията от историята на използване
  - ✓ Оценява се приложимостта му за новото ПО
  - ✓ Специално внимание се обръща на:
    - ✗ версиите на ПО, тяхната поява и същност на промените
    - ✗ документите за описание на грешките при използване и как са били отстранени

Използване на части от ПО

- Провеждат се нови верификационни и нови тестови проверки в случаите на съмнение в достоверността или на отсъствие на резултати за предни проверки.
- Цялостната проверка (старо + ново ПО) след това е като при разработката на ново ПО.

Конфигуриране/Преконфигуриране

- Извършва се в две основни направления:
  - ✓ Валидация на базовото ПО
  - ✓ Валидация на конфигурационните данни.
- Валидацията на базовото ПО се извършва по методите за използване на съществуващо ПО (по един от двата варианта).
  - ✓ За всяка подсистема и за всеки модул, които отговарят за някоя от функциите на новото ПО.
  - ✓ За всяка отделна операция на базовото ПО, които водят до промяна на данните за конфигуриране/преконфигуриране на ПО.
- Валидацията на конфигурационните данни е като за нов програмен продукт.

Валидация на данните

- Валидацията на данните се разделя на две дейности:
  - ✓ Валидация на стойностите на данните
  - ✓ Валидацията на техниките и методите за съхранението, модификацията и използването им.
- И в двата случая насоката на валидацията е да се установи дали данните се използват коректно и дали те позволяват да се създаде ПО, което да функционира коректно.

Валидация на стойностите на данните

- Насоката е да се определи коректността на всяка данна във всеки момент на работа на ПО.
- Съществуват различни форми, като някои са изцяло ръчни, а други изцяло автоматизирани.
  - ✓ Проверка дали входните сигнали се подават в правилните места.
  - ✓ Проверка дали входните сигнали отговарят на критериите за качество и коректност.
  - ✓ Проверка за коректност на реакциите, породени от входните сигнали.
- Препоръчително е първо да се извърши валидация в специално разработена тестова среда, а след това в реалната среда на използване.

Валидация на системата за управление на данните

- Това е валидацията на техниките и методите за съхранението, модификацията и използването им.
- Това се прилага във всички в случаи, независимо дали има автоматизирани средства за управление на БД или няма.
- Подборът на техниките за валидация задължително зависи от типа на съхраняваната информация и нейното използване.
  - ✓ Проверяват се само тези елементи, които оказват влияние върху функционирането на конкретното ПО.
  - ✓ За автоматизирани системи за управление на БД най-често се използват изискванията на стандарта IEC 880.

Като част от система

- Особености
  - ✓ В този случай част освен дейностите по валидация на програмното осигуряване самостоятелно се провеждат дейности по валидация на системата като цяло.
  - ✓ Задължително се оценява връзката ‘Изисквания към изискванията – Изисквания към ПО’
  - ✓ При някои случаи на разработка (използване на части от ПО, конфигурация/реконфигурация) може директно да се извърши валидация на ниво система.

Като система за производство на продукти

- Валидацията е резултат от успешната валидация на следните елементи:
  - ✓ Оценява се както като част от система.
  - ✓ Валидация като част от технологичен процес.
  - ✓ Валидация на произведения продукт.

Въпроси ?

Валидация и верификация на вградени системи

Вградена система

- Дефиниция
  - ✓ “Това е компютърна система, която е създадена като компонент на по-голяма система и е ориентиран към работа върху един процесор. Задачите на тази система са много по-различни от системите с общо предназначение.”
  - ✓ “… устройство, което има компютърна логика в ИС и не може да бъде директно програмирано от потребителя. Захранва се от батерия или директно от мрежата. ИС управлява една или повече функции на устройството.”

Вградена система

- Дефиниция
  - “Комбинация от апаратна част и програма, заедно с допълнителни механична и/или други части (ако е необходимо), използвани за постигане на определена функционалност.”
  - “Вградените системи са компоненти, интегриращи апаратна и програмна част, специално разработване за свързване и реализация на специфична задача.”

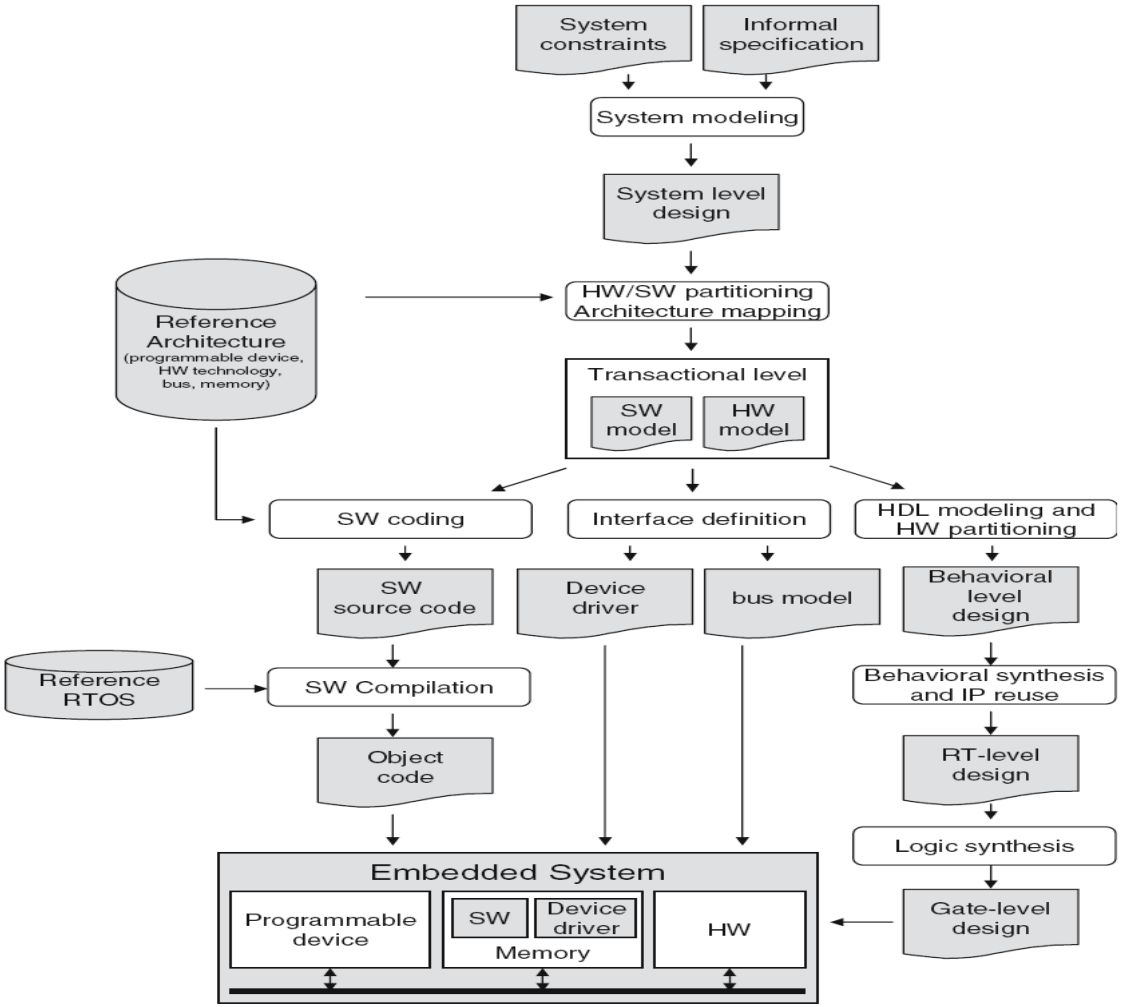
Вградена система

- Особености
  - Всяка от отделните части (апаратна и програмна) не може самостоятелно да осигури нужната функционалност.
  - Основни ограничения
    - Реактивни системи
    - Най-често ‘критични’ системи
    - Автономни системи
    - С високо ниво на сигурност и надеждност
    - Много често имат ясно дефинирани ограничения по време.

Вградена система

- Особености
  - Ясно дефинирани промени в разбиранията за ‘ограничения’ между различните приложения и фамилии.
    - Силиконови системи (system-on-chip)
    - Класически системи
    - Други
- Валидация и верификация
  - Верификация на различните етапи на разработка
  - Валидация
    - на модела, на комуникациите, на работата и функционална валидация

Верификация на модела на изграждане



Източник:
Dynamic and FormalVerification of Embedded Systems:A Comparative Survey

Верификация на модела на изграждане

- Нива в дизайна на вградената система
  - ✓ ‘Система’
  - ✓ ‘Архитектура’
  - ✓ ‘Програмен код’
  - ✓ ‘Изпълним код’
  - ✓ ‘Дефиниция на интерфейси’
  - ✓ ‘Синтез на поведение’
  - ✓ ‘Логически синтез’

Ниво на дизайн: ‘Система’

- Няма разлика между апаратна и програмна част, т.е. методите за валидация и верификация оценяват системата като цяло.
- Не се разглеждат комуникационни протоколи между блоковете, анализ на закъснението на разпространение на сигналите, изчислителни алгоритми.
- Оценяват се гъвкавостта на дизайна, ефективността и функционалността му.

Ниво на дизайн: ‘Архитектура’

- Оценява се обвързването на програмна и апаратна функционалност (на ниво блокове).
  - ✓ Модел на паметта
  - ✓ Комуникационната архитектура между програмна и апаратна част и топология на шината
  - ✓ Програмируемите устройства, върху които ще се изпълнява програмната част
  - ✓ Апаратната технология, която се използва за реализация на апаратните задачи.
- Оценка на транзакционността - поради отделеността на комуникация от изчисление се използват методи за симулация.

Ниво на дизайн: ‘Програмен код’

- Основно се оценяват проблемите, свързани с ограниченията на устройствата, върху които ще се изпълняват програмния код и ограниченията, резултат от комуникационния интерфейс.
  - ✓ Апаратната част участва в процеса като ‘черна кутия’.
  - ✓ Комуникацията между апаратна и програмна част е не основата на драйвери.
- Оценява се проблемите от избора на език за програмиране.

Ниво на дизайн: ‘Изпълним код’

- Оценява се изборът на решение за изпълнение на програмния код:
  - ✓ С използването на ОСРВ (операционна системи за реално време) или директно управление на апаратната част
  - ✓ Компилативен или интерпретативен вариант.
  - ✓ Зависимост от компютърната платформа.

Ниво на дизайн: ’Интерфейсите’

- Разделянето на функциите на системата, решавани с апаратни средства между отделни модули, налага реализация и оценка на интерфейсите между тези модули.
  - ✓ Времевият модел за апаратната и за програмната част е различен (обикновено събитийно базиран – циклично изпълняван) , т.е. трябва преобразуване в двете посоки.
- Оценка на драйверите за управление на устройствата и на протокола на шината
  - ✓ функционална прозрачност на апаратната част спрямо програмната
  - ✓ надеждност, устойчивост, пропускателна способност

Динамична верификация

- Оценява коректността на дизайна на основата на симулативни техники
  - ✓ Оценява се реакцията на системата на множество входни стимули, генерирани от симулаторната среда.
  - ✓ Тестовите набори, създадени за по-високо абстрактно ниво се използват и за по-ниските нива, като непрекъсната се добавят и нови.
- Използва се за определяне на наличието на дефекти, но не може да гарантира отсъствието им.

Динамична верификация

- Може да се използва както за системата като цяло и поотделно за апаратната и за програмната част.
- Необходими елементи за оценката на системата
  - ✓ Симулационен модел на дизайна на системата
  - ✓ Симулатор
  - ✓ Тестова среда
  - ✓ Метод за оценка на коректността на резултатите.
    - ✗ Това е основният проблем поради невъзможност за реализиране на изчерпателен тест при симулация.

Динамична верификация

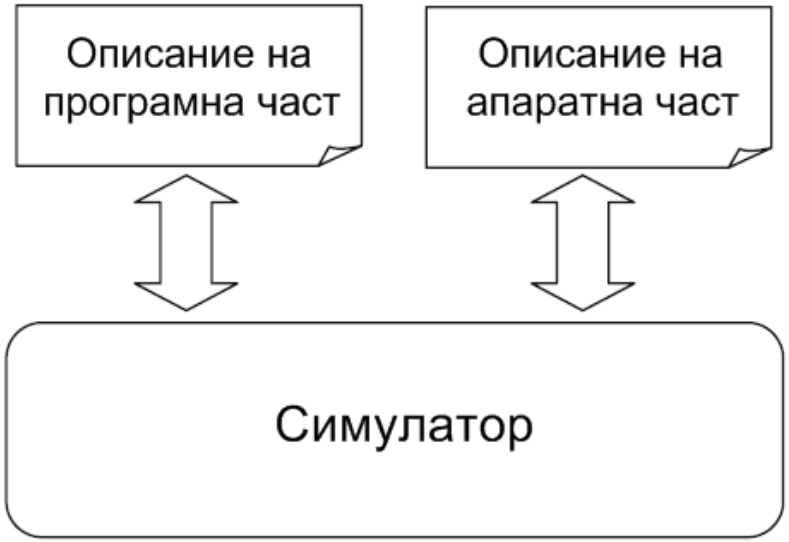
- Според метода за оценка на коректността на резултатите съществуват два вида динамична верификация.
  - Симулация на логиката на системата
    - Качеството се определя възоснова на покритие на кода (оператор, разклонение, път, цикъл, условие и т.н.)
  - Симулация на дефектите
    - Качеството се оценява чрез анализ на отношението между открити дефекти и реално дефекти – дефектите се симулират (различни видове и модели) и се оценяват резултатите в с/бездефектна система.
    - 100 % не гарантират пълно управление на дефектите, а само 100 % откриваемост за симулираните условия.

Ко-симулативни подходи

- Ко-симулативния подход се използва за оценяване на системата след разделянето на функционалността между програмната и апаратната част.
- Използват се специално разработени платформи, различаващи се по архитектурна насоченост, ефективност на изпълнението, езици за описание на системата
  - ✓ Хомогенни среди
  - ✓ Хетерогенни среди
  - ✓ Полу-хомогенни среди

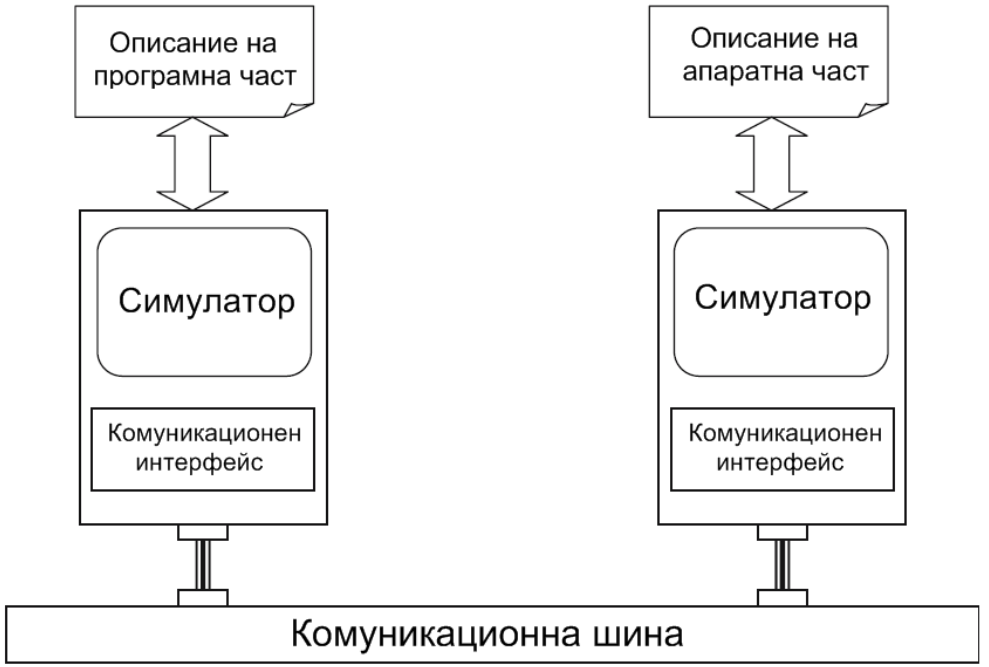
Хомогенни среди

- За симулацията на апаратна и програмна част се използва едно общо ядро.
- Използва се абстракция от висок ред за представяне на разликата между апаратни и програмни блокове
- Използва се за
  - ✓ оценка на дизайн
  - ✓ оценка на използване
- Недостатък
  - ✓ Реализация на много различни средства за апаратна и програмна реализация



Хетерогенни среди

- Много по-точна реализация на поведението на апаратната и програмната част.
  - ✓ Използва се по-ниско ниво на абстракция за апаратната част.
  - ✓ Програмата част се оценява в изпълним вид.
- Същност
  - ✓ Изпълняват синхронизация между два и повече симулатора.



Хетерогенни среди

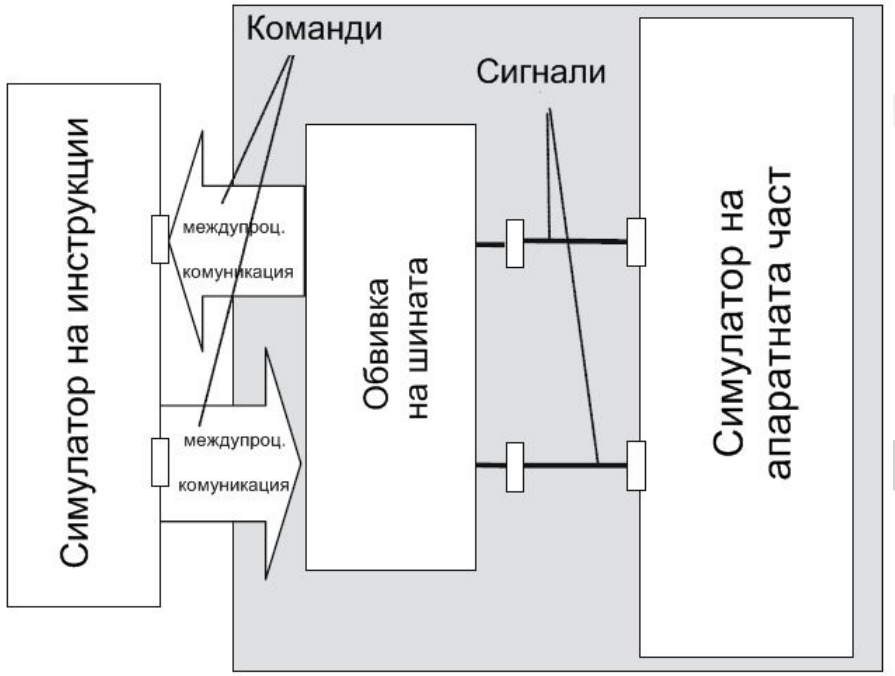
- Основен проблем:
  - Ефективността на симулацията на връзката между събитийно-ориентираната апаратна част и симулатора на инструкции, необходим за програмната част
    - Необходим е симулатор на комуникационния канал (шината) за връзка ‘апаратни елементи-модули за изпълнение на програмата’ – моделиране на сигналите.
- Много бавна симулация.
  - Симулацията на всички сигнали по шината съществено намалява скоростта.

Полу-хомогенни среди

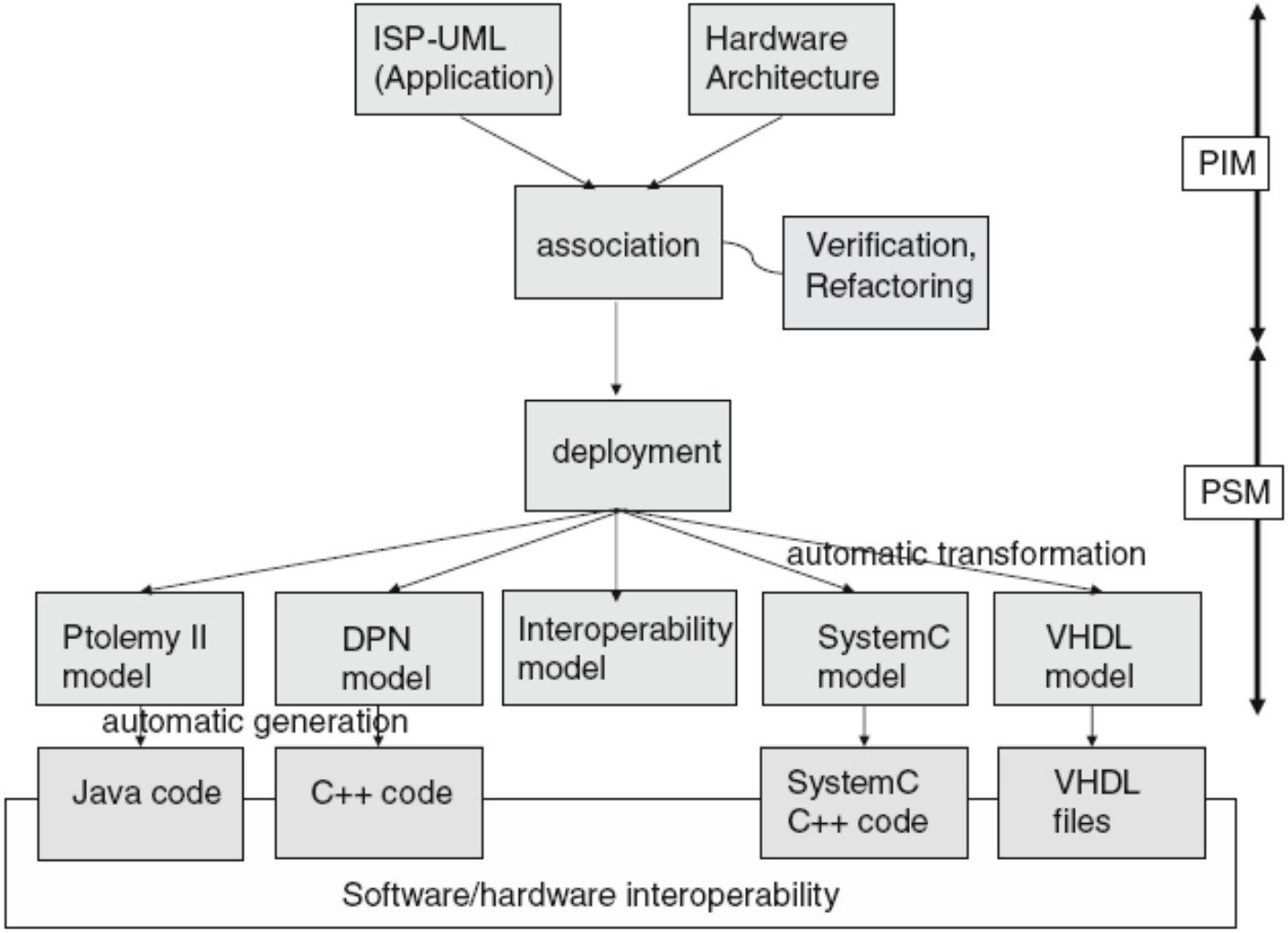
- Същност
  - При тях няма симулация на шината на ниско ниво.
    - Реализира се чрез малък брой функции, предаващи информацията за нужното време без да симулират всички сигнали.
  - Хомогенни от гледна точка на използвания програмния език и хетерогенни от гледна точка на симулацията.
- По-ниска точност от хетерогенните среди, но скоростта на симулацията е по-висока.

Полу-хомогенни среди

- Две основни системи
  - ✓ Междупроцесна комуникация
  - ✓ Обвивка на шината
    - Програмно абстрактно ниво
  - ✓ Най-често интерфейсът между обвивката на шината и симулатора на инструкции е променяем



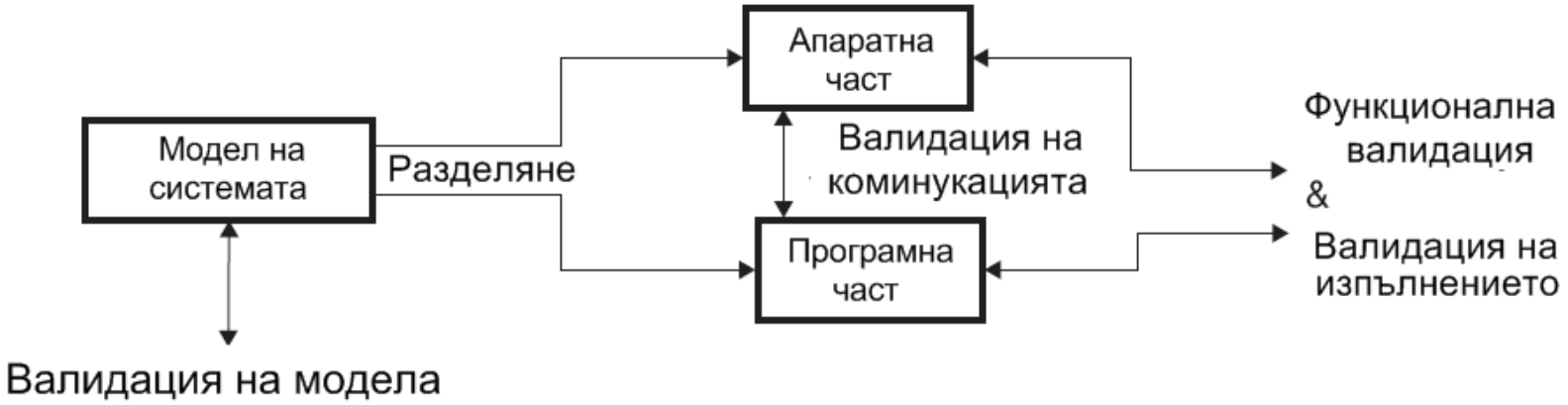
UML-базирана верификация



Източник:
Dynamic and FormalVerification of Embedded Systems:A Comparative Survey

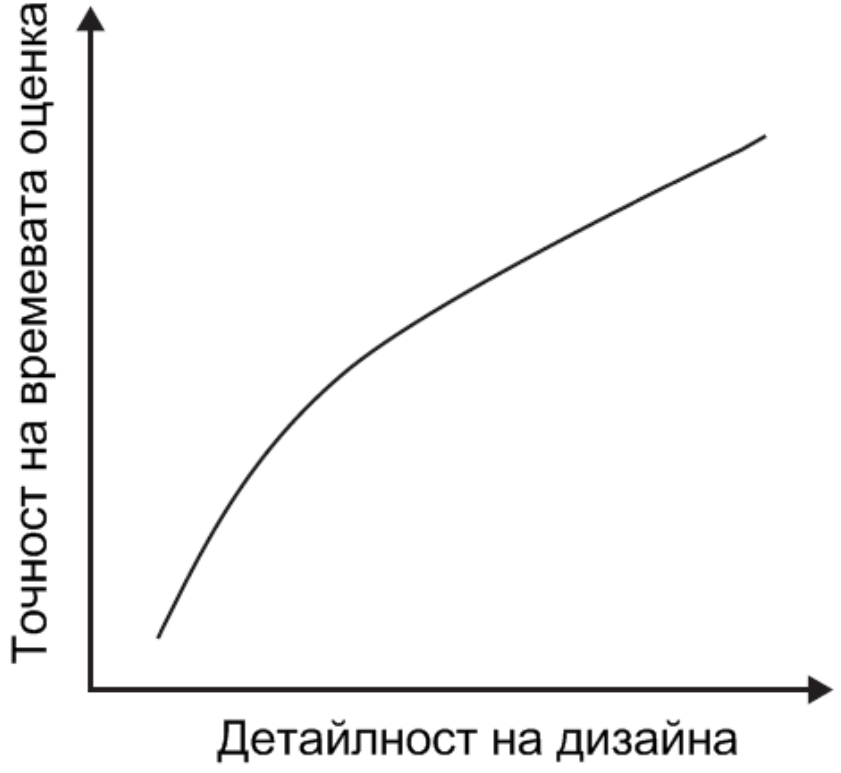
Валидация на вградени системи

- Валидация на модела
- Валидация на реализацията
  - ✓ Валидация на Високо ниво на реализация: комуникация между блоковете
  - ✓ Валидация на ниско ниво
    - × Валидация на изпълнението
    - × Функционална валидация



Валидация на вградени системи

- Проблеми:



Валидация на модела

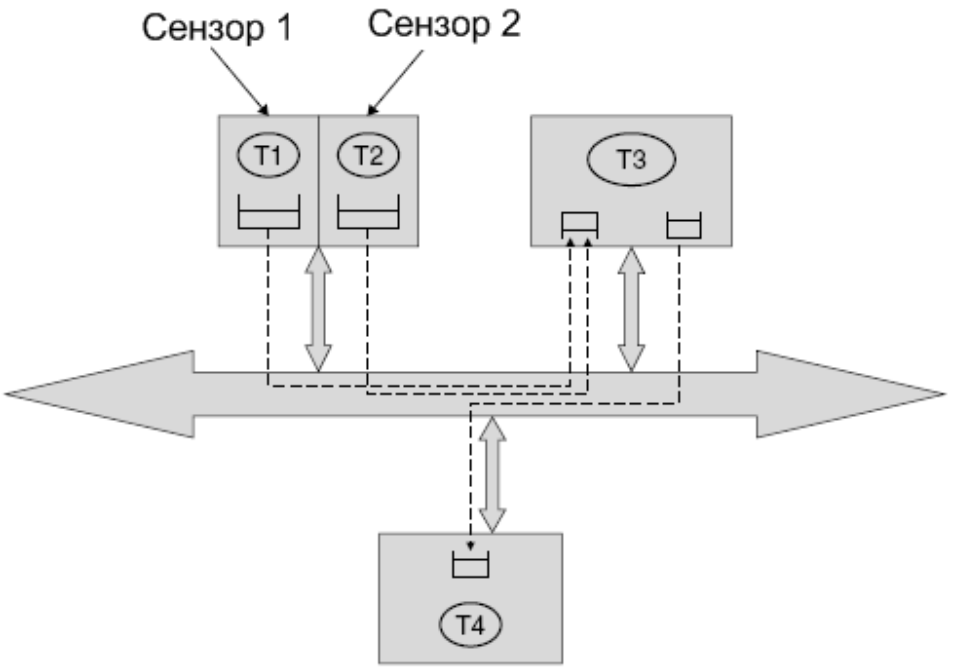
- Общо описание на модела
  - ✓ Крайни автомати (Finite-State Machines - FSMs)
  - ✓ Комуникиращи си крайни автомати (Communicating FSMs)
  - ✓ Базирани на графици от съобщения (Message Sequence Chart–Based Models)
- Симулационни модели
  - ✓ Симулация на крайни автомати
  - ✓ Симулация на модели, базирани на графици от съобщения
- Моделно-базирано тестване
- Проверка на модела
  - ✓ Проверка на спецификацията
  - ✓ Проверка на процедурно ниво

Валидация на модела

- Основен проблем
  - Разликата между разработваната система и платформата, върху която ще се имплементира
    - “Моделът на поведение представлява описание на механизъма на промяна на състоянията при комуникация между обекти.”
  - При вградените системи има много силна обвързаност между платформа и разработвана система.
    - Валидация на платформения модел
    - Валидация на системния модел на основата на резултатите за платформения модел

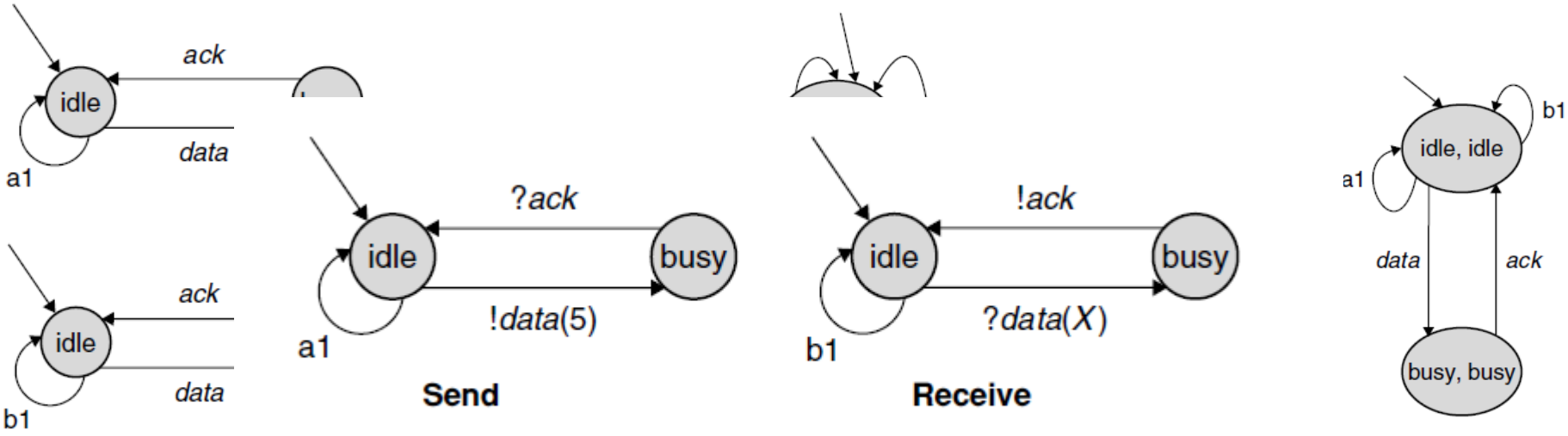
Валидация на модела

- Основен проблем
  - ✓ Функционални изисквания
  - ✓ Изисквания за времеви ограничения



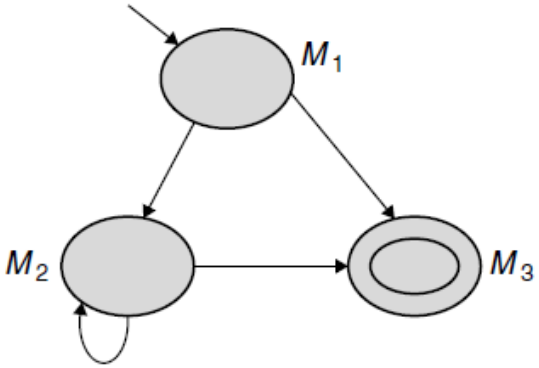
Валидация на модела

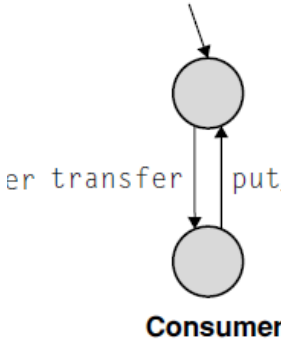
- Общо описание на модела
  - ✓ Крайни автомати (Finite-State Machines - FSMs)
  - ✓ Комуникиращи си крайни автомати (Communicating FSMs)
  - ✓ Базирани на графици от съобщения (Message Sequence Chart–Based Models)

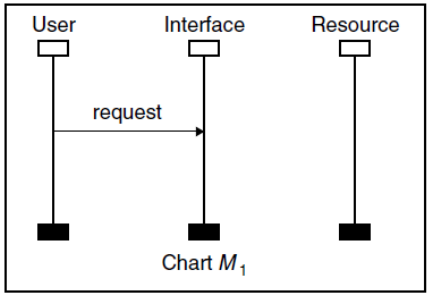


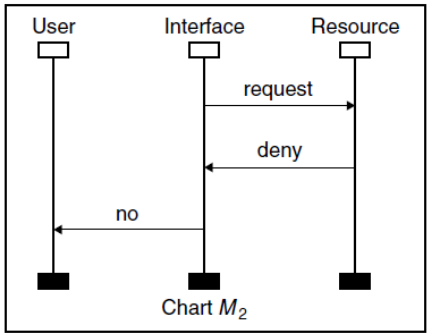
Валидация на модела

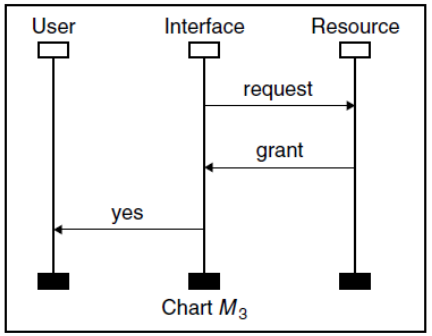
- Общо описание на модела
  - ✓ Крайни автомати (Finite-State Machines - FSMs)
  - ✓ Комуникиращи си крайни автомати (Communicating FSMs)
  - ✓ Базирани на графици от съобщения (Message Sequence Chart–Based Models)





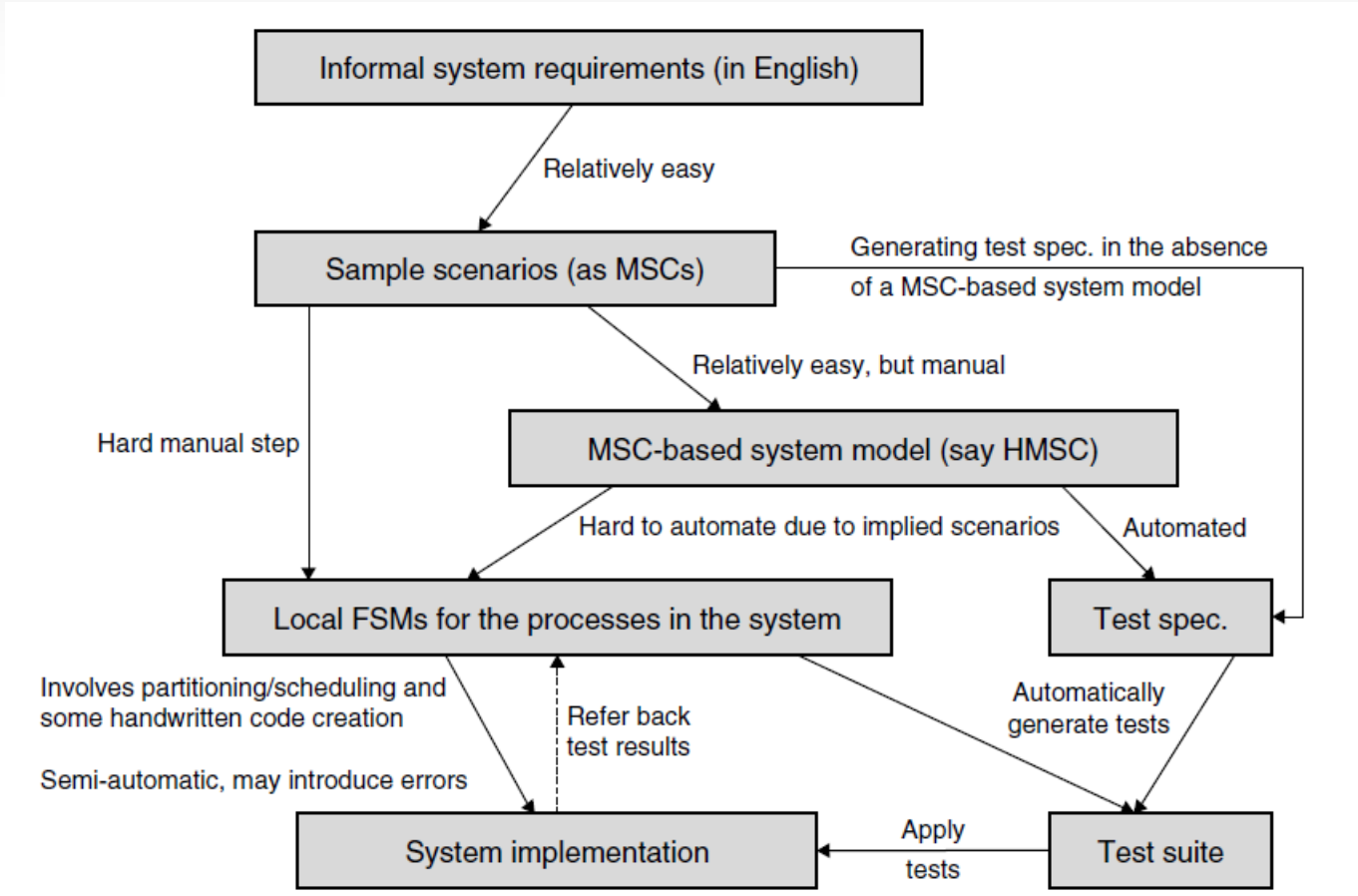




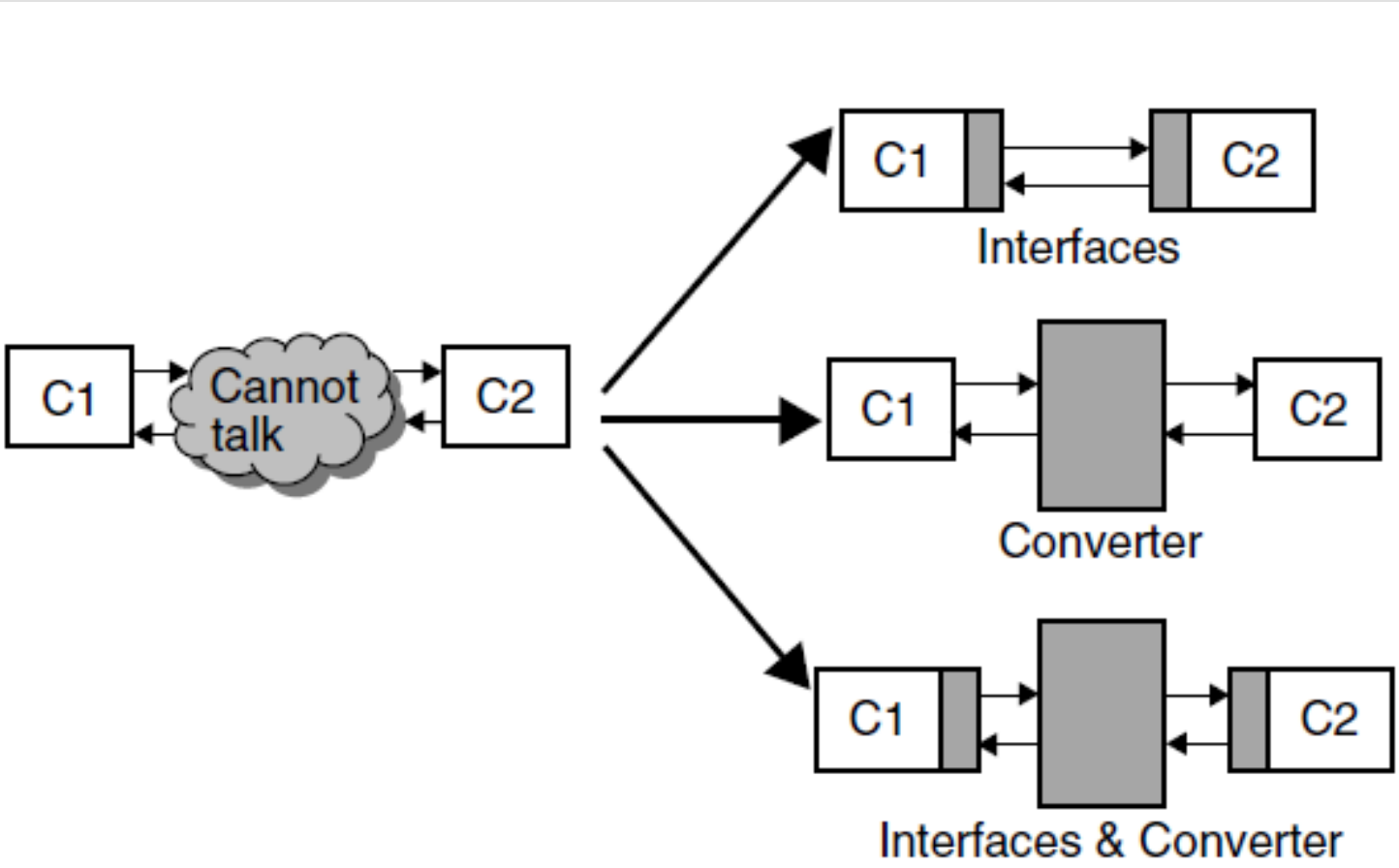


Валидация на модела

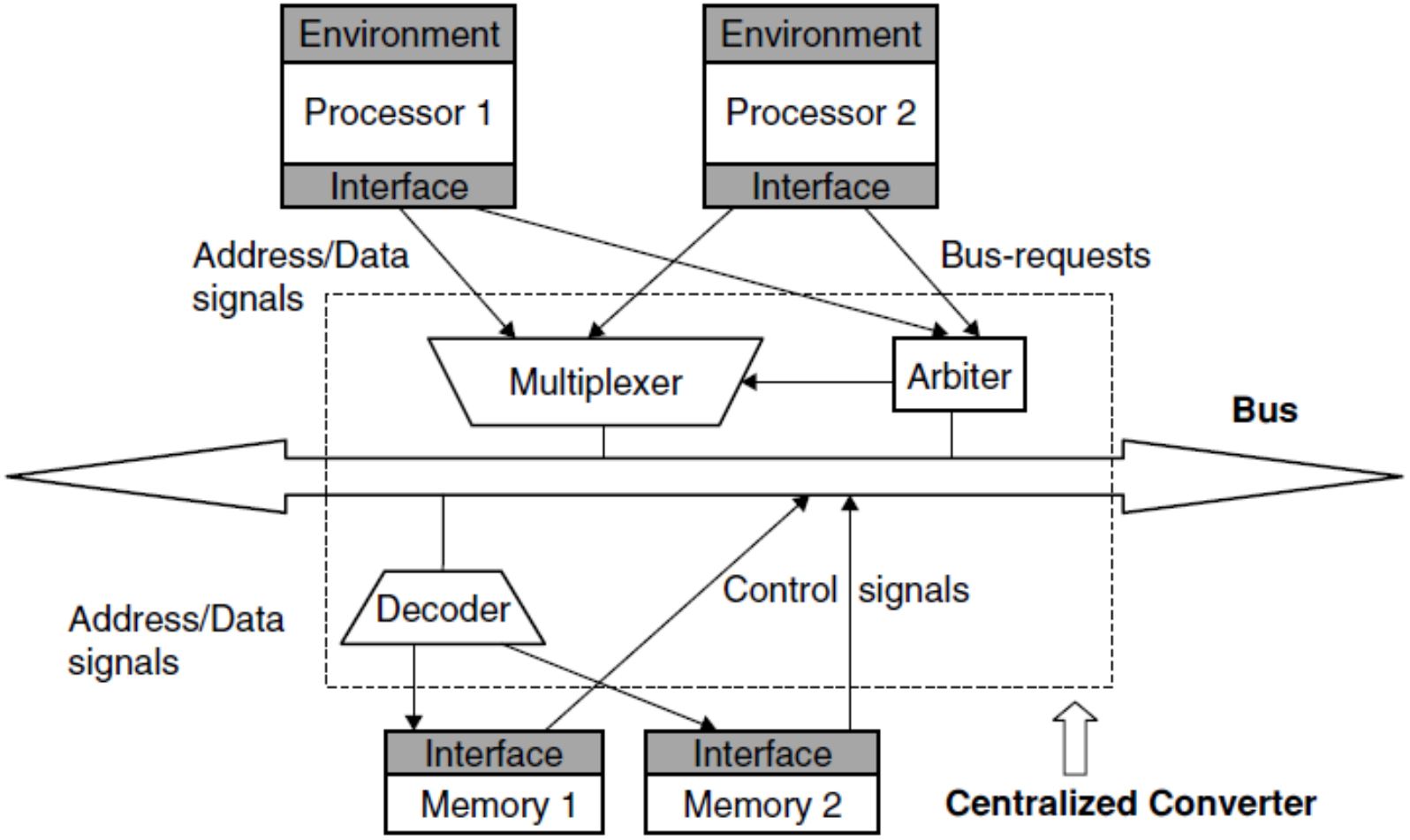
- UML-базирано моделиране



Валидация на комуникацията



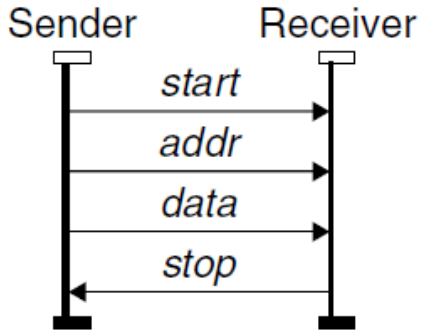
Валидация на комуникацията



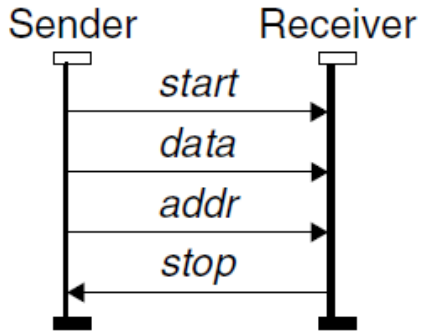
Валидация на комуникацията

- Обща несъвместимост
  - ✓ Приемане/Предаване на сигналите в различен ред
    - × Еднакви сигнали, но предавателя има различна задължителна последователност от приемника
  - ✓ Използване на различно множество от сигнали
  - ✓ Разлика във формат на данните
    - × Например предаване по битове, а приемане на бяйтове
  - ✓ Объркване на данните
    - × Най-често данни с адреси
    - × Данни от различни пакети

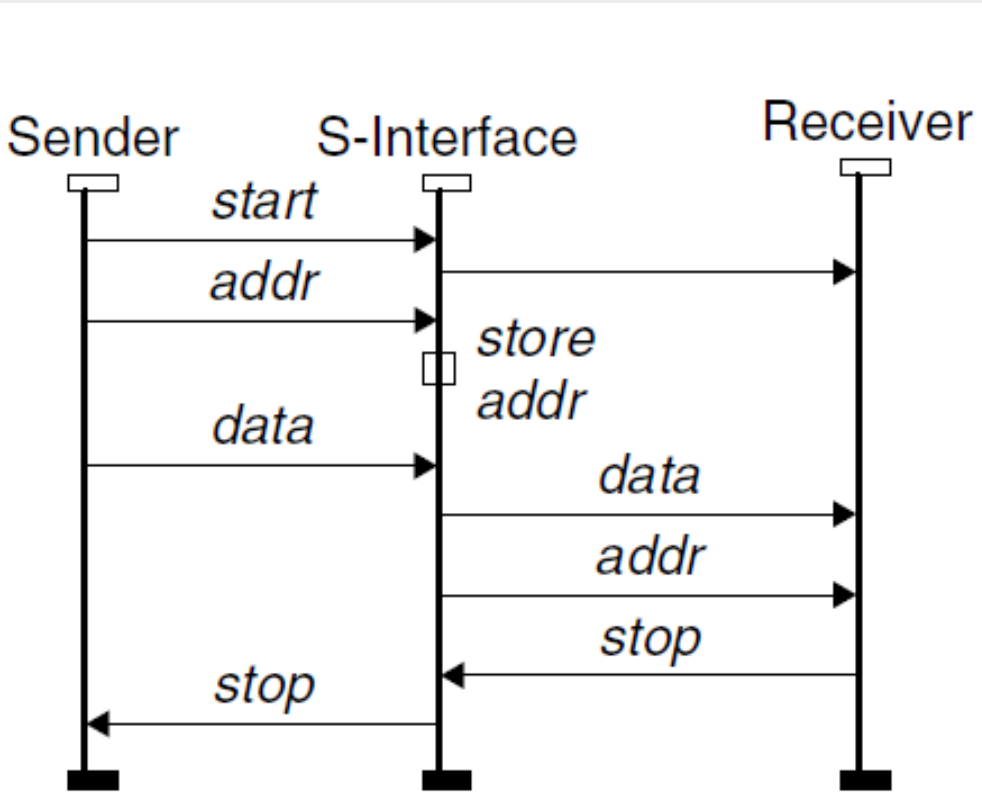
Приемане/Предаване на сигналите в различен ред



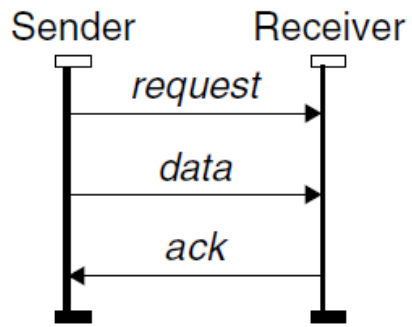
Sender ->



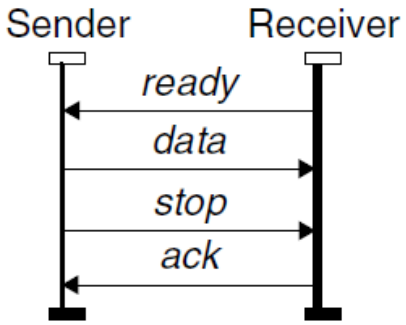
Receiver ->



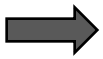
Различно множество от сигнали

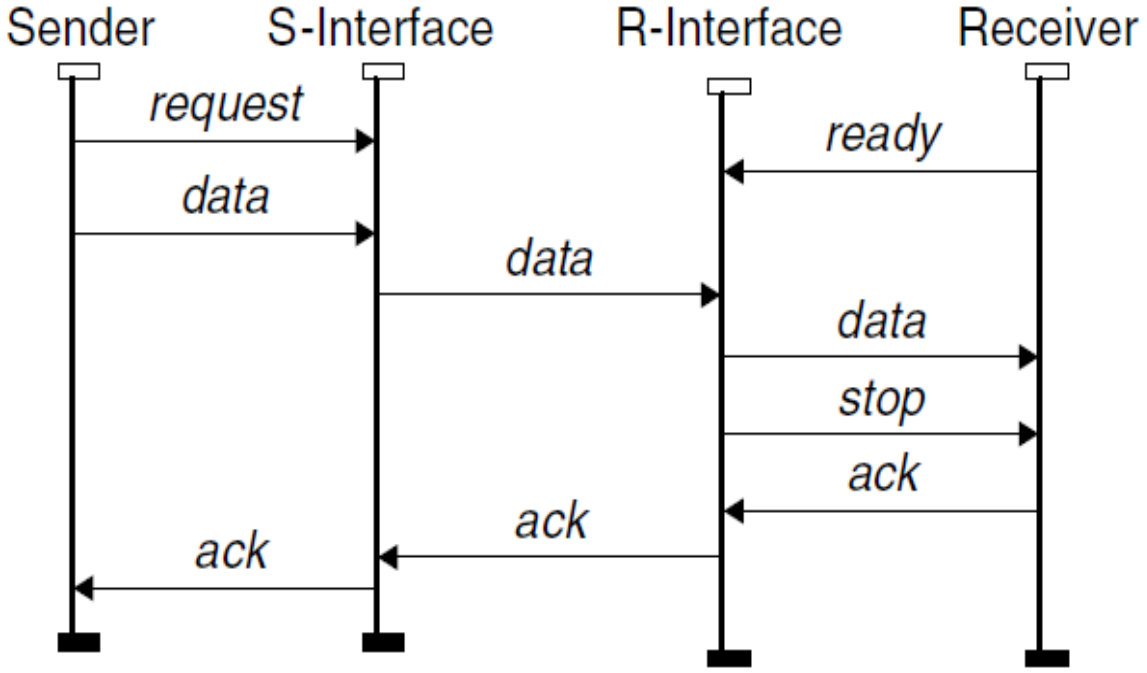


Sender ->



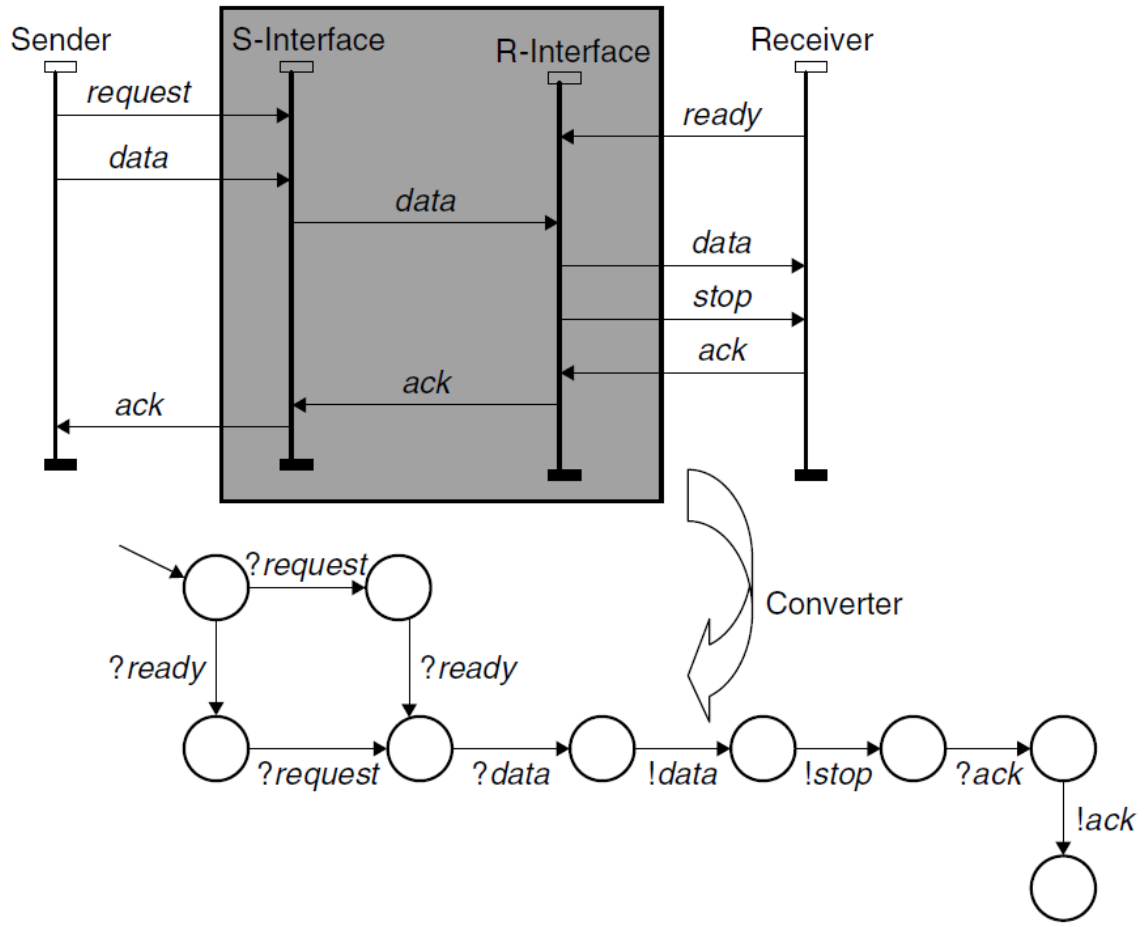
Receiver ->



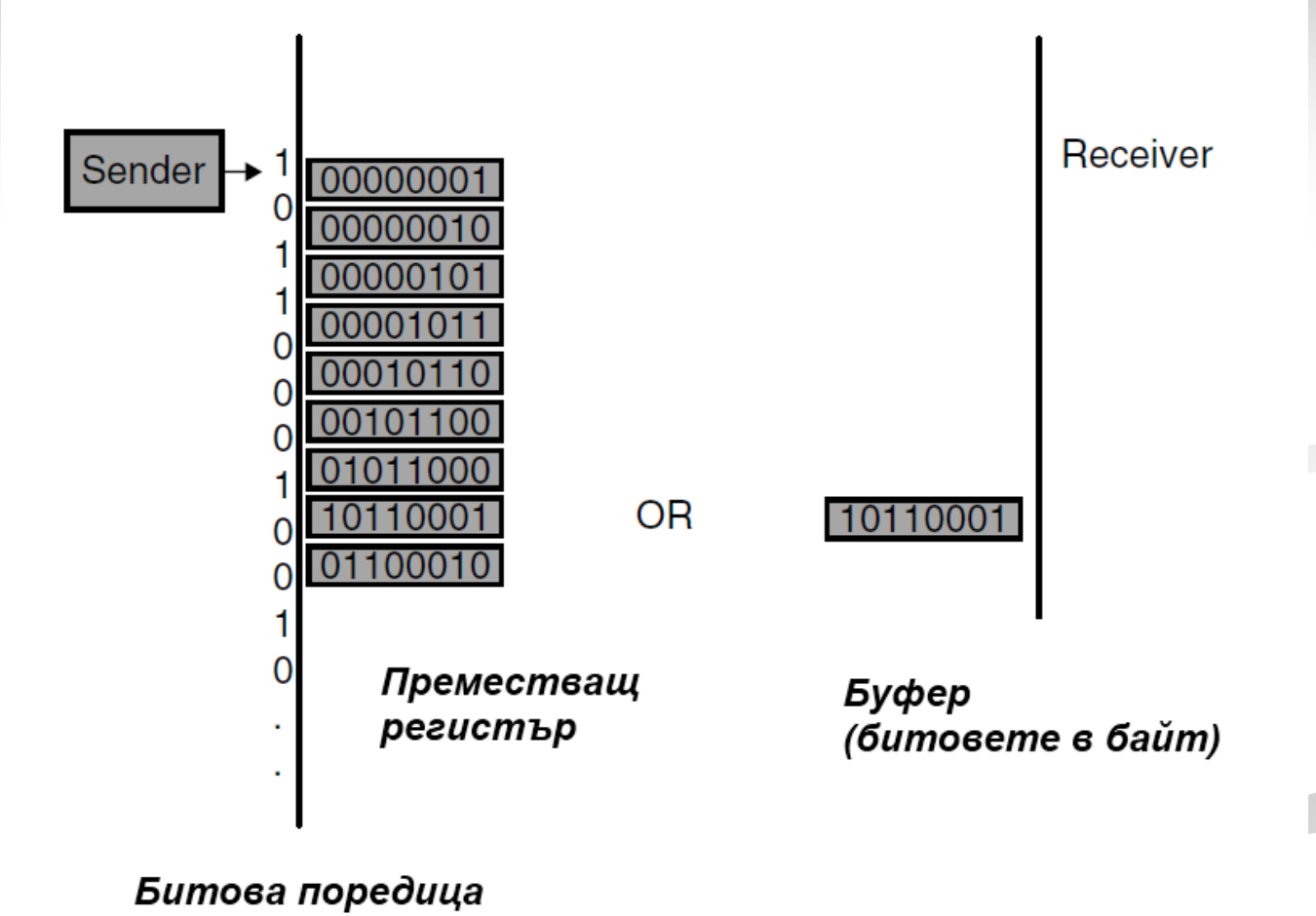


Различно множество от сигнали

- Централизиран конвертор на сигнали

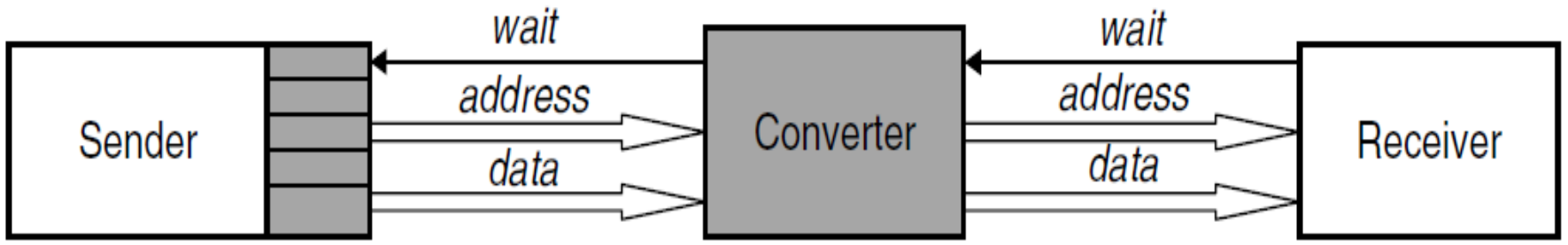


Разлика във формат на данните



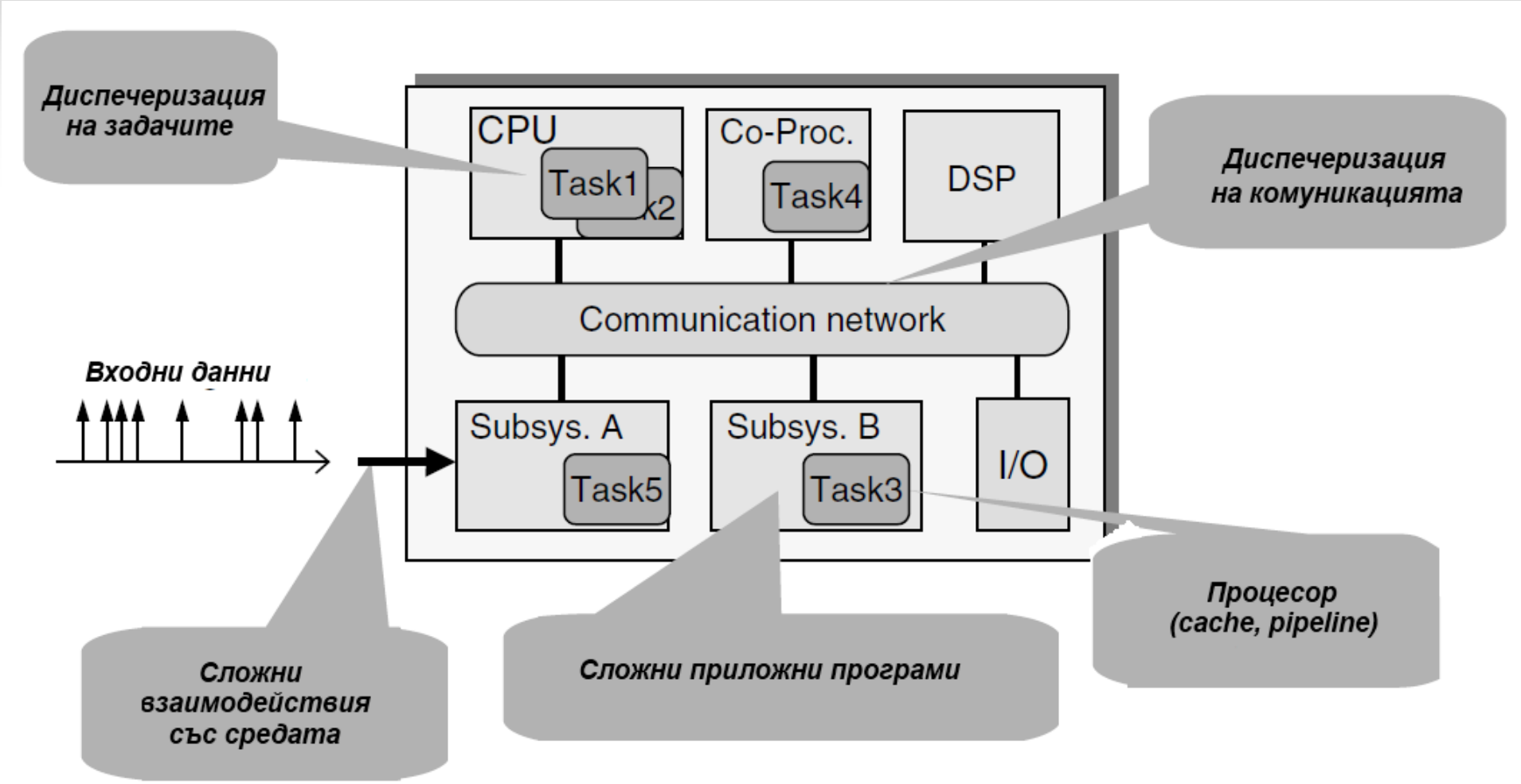
Объркване на данните

- Синхронизационни проблеми

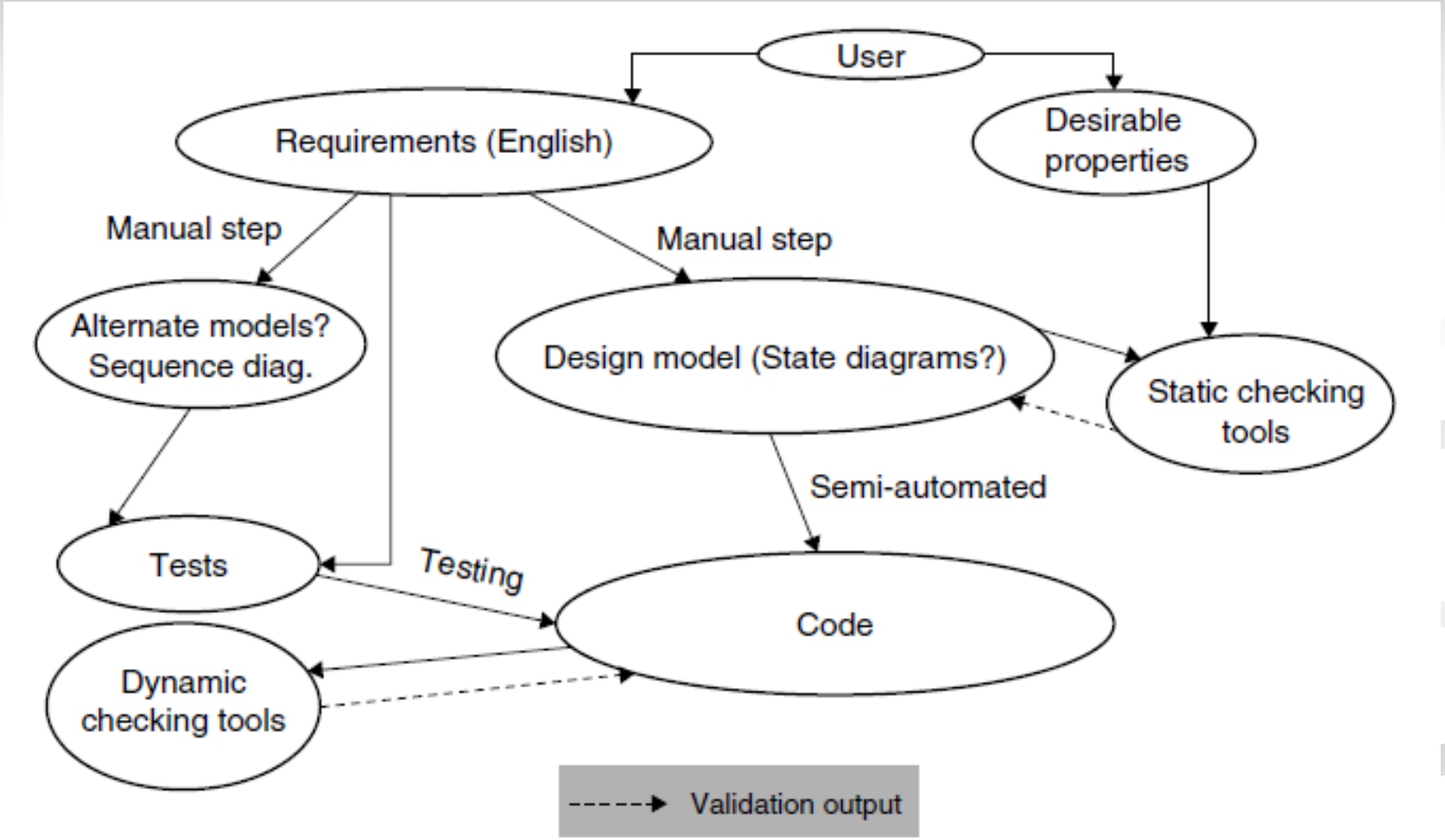


Буфер за съхранение на addr/data когато wait е вдигнат

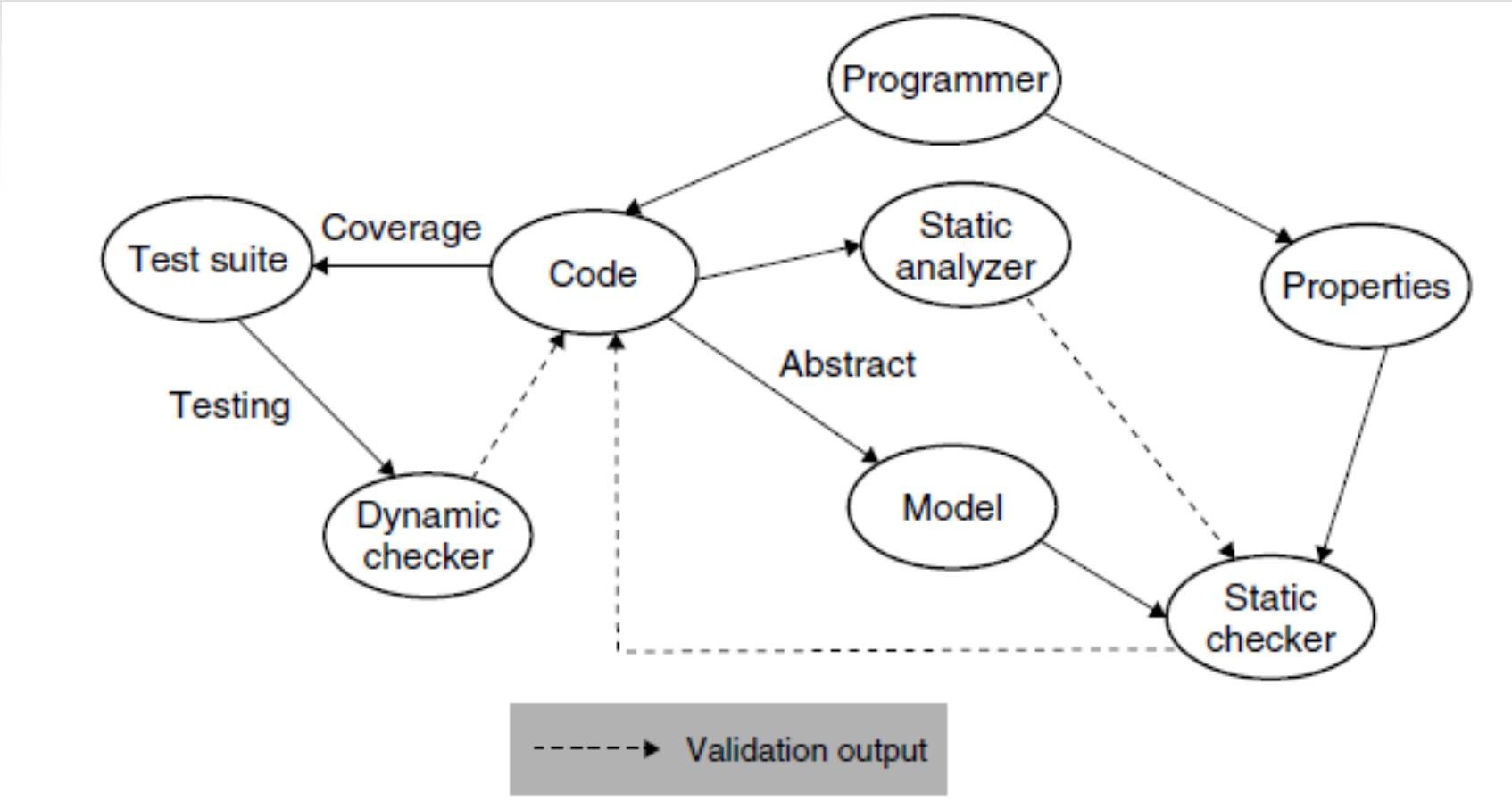
Валидация на изпълнението



Функционална валидация



Функционална валидация



Въпроси ?

Валидация и Верификация на Програмни проекти

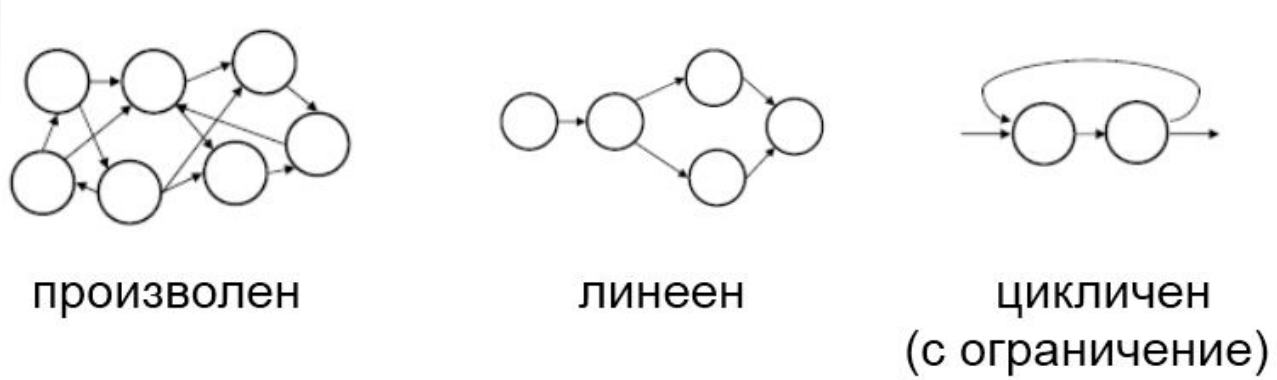
Особености при ссновни архитектурни стилове

Data Flow

- Същност:
  - ✓ Данните контролират изчисленията
    - ✗ Структурата на дизайна се определя от нормалното движение на данни от компонент на компонент
  - ✓ Моделът на потока от данни е зададен имплицитно
    - ✗ Това е единствената форма на комуникация между компонентите
- Възможни варианти:
  - ✓ Как се управлява контрола – зареждане или извличане (push versus pull)
  - ✓ Степен на паралелна обработка (конкурентни процеси)
  - ✓ Топология

Data Flow

- Модели на потоци от данни



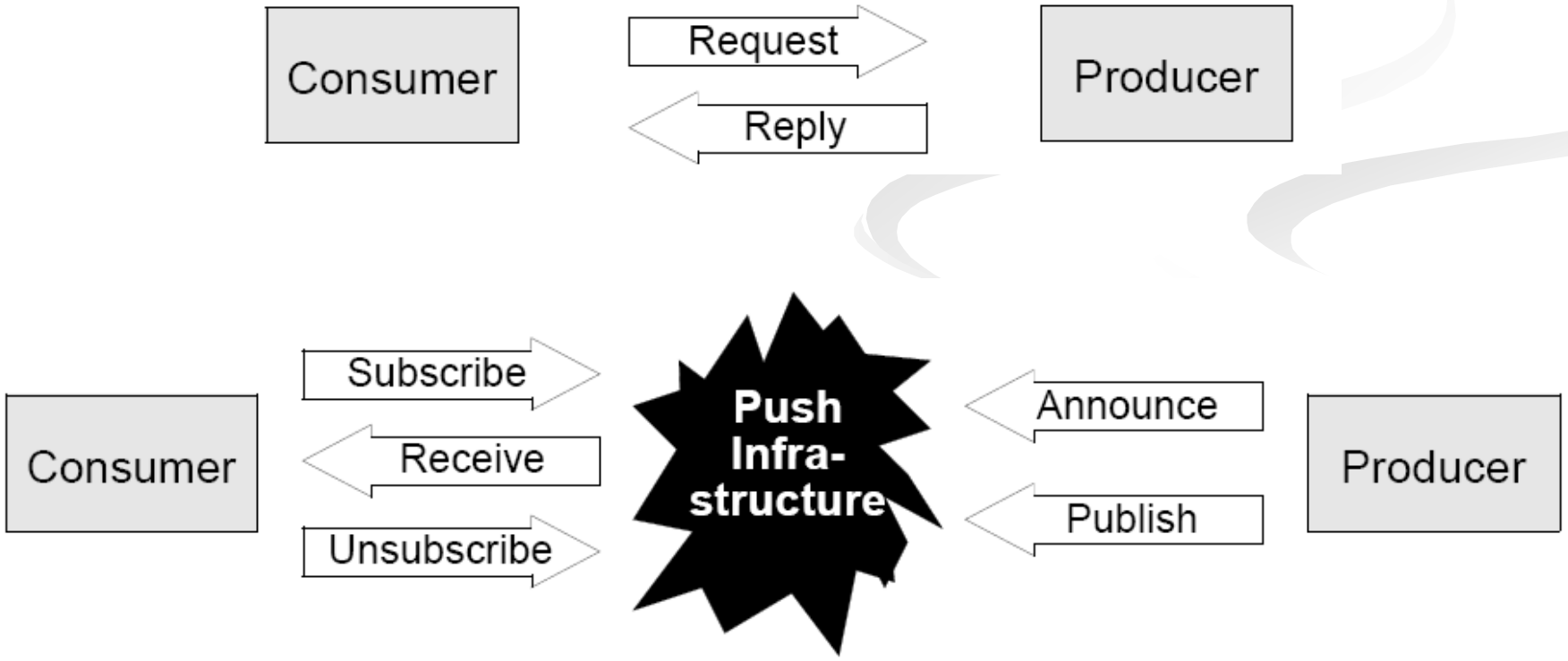
- Видове архитектури
  - ✓ Пакетна обработка (Batch sequential)
  - ✓ Потокови мрежи (pipes&filters)
  - ✓ Архитектури със затворен цикъл на управление

Data Flow

- Елементи
  - Компоненти
    - Интерфейсът им е входни и изходни портове
    - Изчислителен модел - *Четене от входен порт; Изчисление; Запис в изходен порт*
  - Свързващи елементи (конектори)
    - Еднопосочни, най-често асинхронни и буферирани
    - Изчислителен модел - *Транспорт от изходен порт до входен порт*
  - Системи
    - Произволни графи
    - Изчислителен модел - *Функционална композиция*

Push-Based Style

- Същност:
  - ✓ Класическа задача ‘Производител-Потребител’



Push-Based Style

- Различава се съществено от базираните на извличане (pull-based), например Web-приложенията.
- Елементи
  - ✓ Компоненти – производител/-и и потребител/-и
  - ✓ Конектори – канали, разпространител (broadcaster), транспортна система, повторители, кеш, прокси
- Асиметричен комуникационен модел
  - ✓ Производителите не са приемници.
  - ✓ Може да има неколко производителя и всеки да има няколко потребителя (много повече, спрямо архитектурите от вида '*Event-based*')
- Основени проблеми
  - ✓ Състезание на сигнали (race condition)
  - ✓ Взаимна блокировка (deadlock)

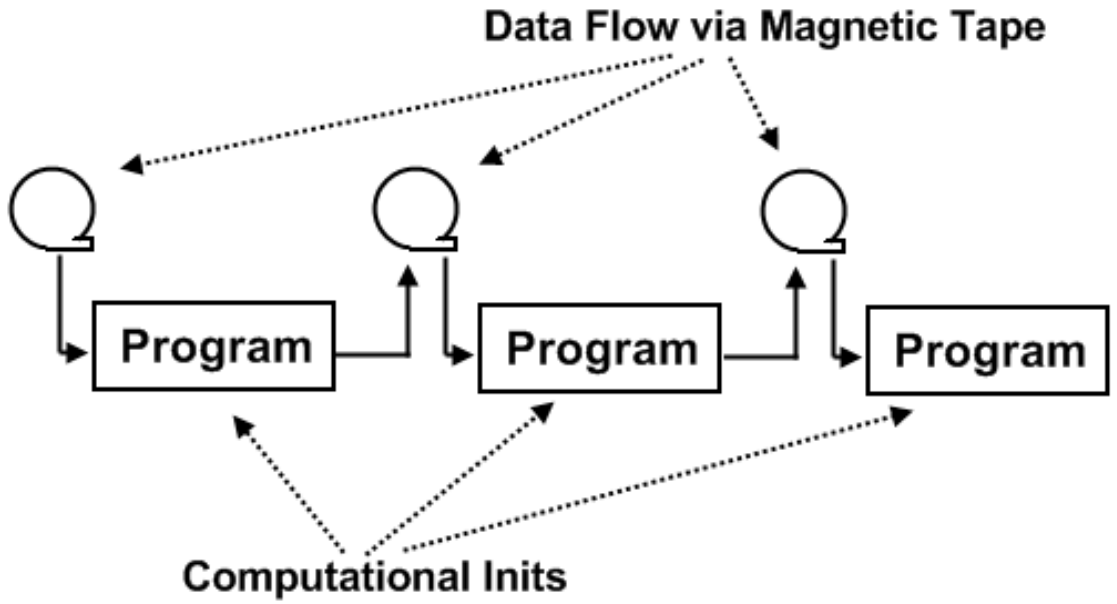
Push-Based Style

```
int itemCount = 0;

procedure producer() {
    while (true) {
        item = produceItem();

        if (itemCount == BUFFER_SIZE) {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}
```

```
procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

Пакетна обработка (Batch Sequential)

- Компонентите са независими програми
- Конекторите са някаква форма на медия
- Всяка програма завършва напълно работата си преди да започне следващата

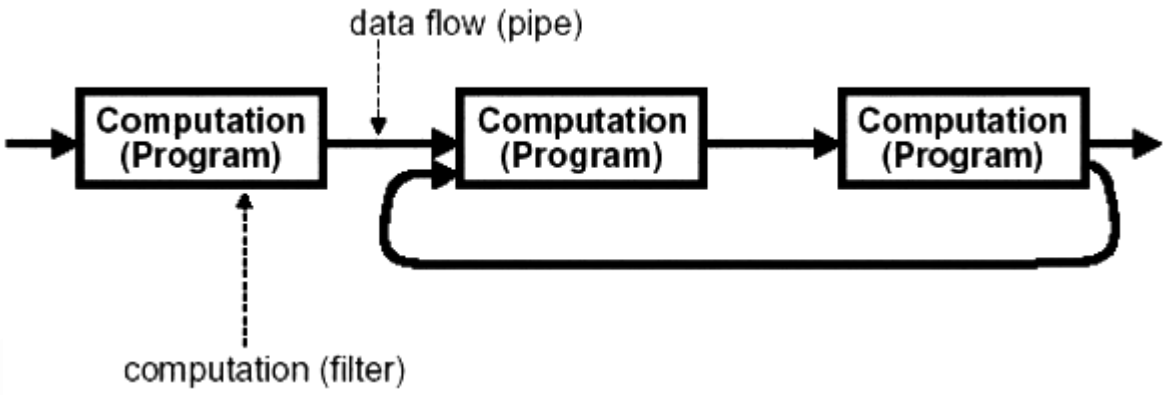


Пакетна обработка

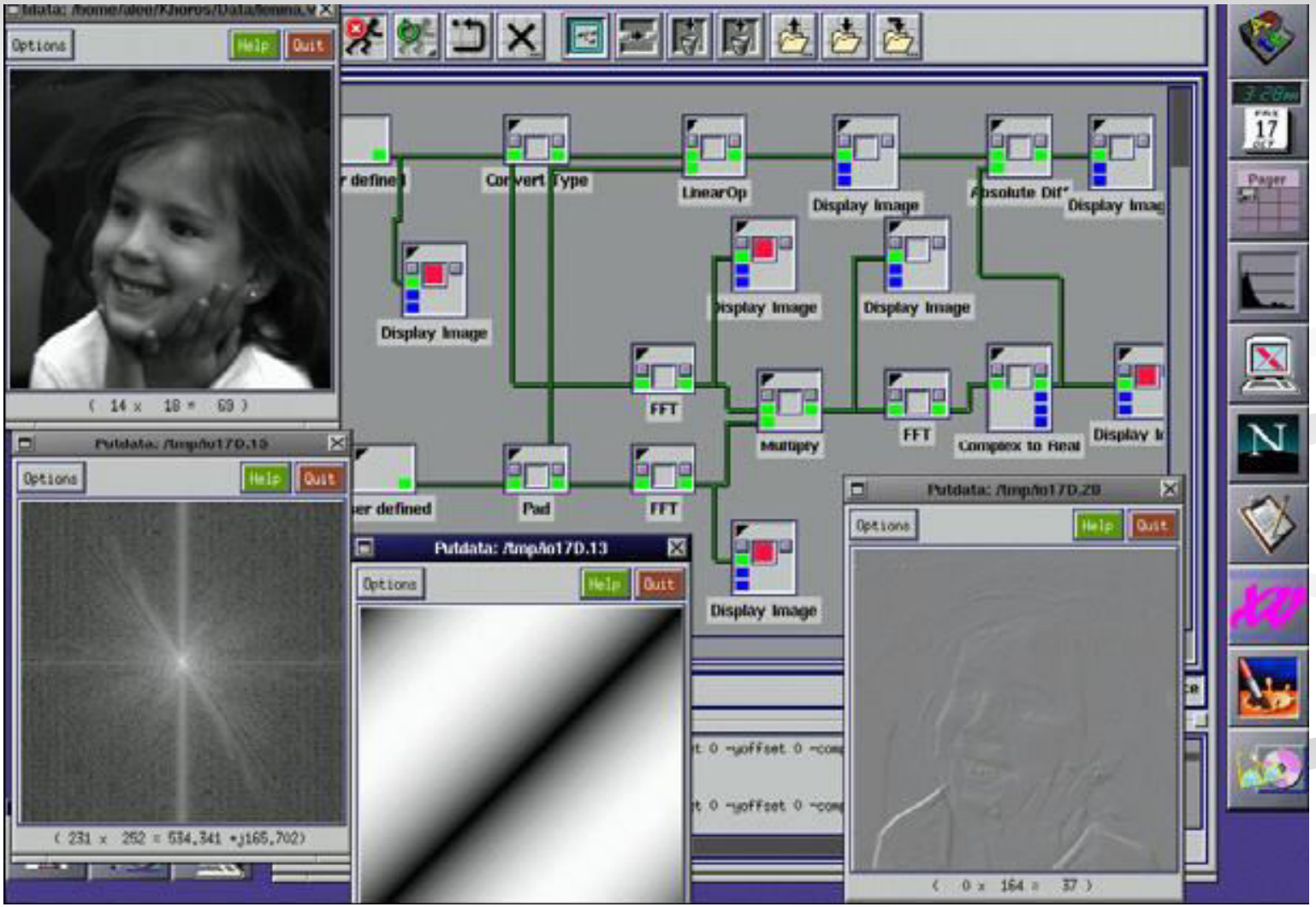
- Особености
  - Лимитиран обем на преносната медия, напр. дисково пространство.
  - Блокова система на диспечиране на процесорното време
  - Не реално-времеви приложения, ориентирани към пакетна обработка, т.е. използване за бизнес приложения:
    - Дискретни транзакции на предварително определен тип данни на периодични интервали
    - Създаване на периодични отчети, въз основа на периодично актуализиране на данни данни

Pipes and Filters

- Това са системи за потокова обработка на информацията.
  - ✓ Данните се обработват инкрементално.
  - ✓ Всяка стъпка на обработка е реализирана с отделен компонент (filter), като тяхната рекомбинация позволява изграждането на фамилия от системи.
  - ✓ Изискват език за програмиране и ОС.



Pipes and Filters



Pipes and Filters

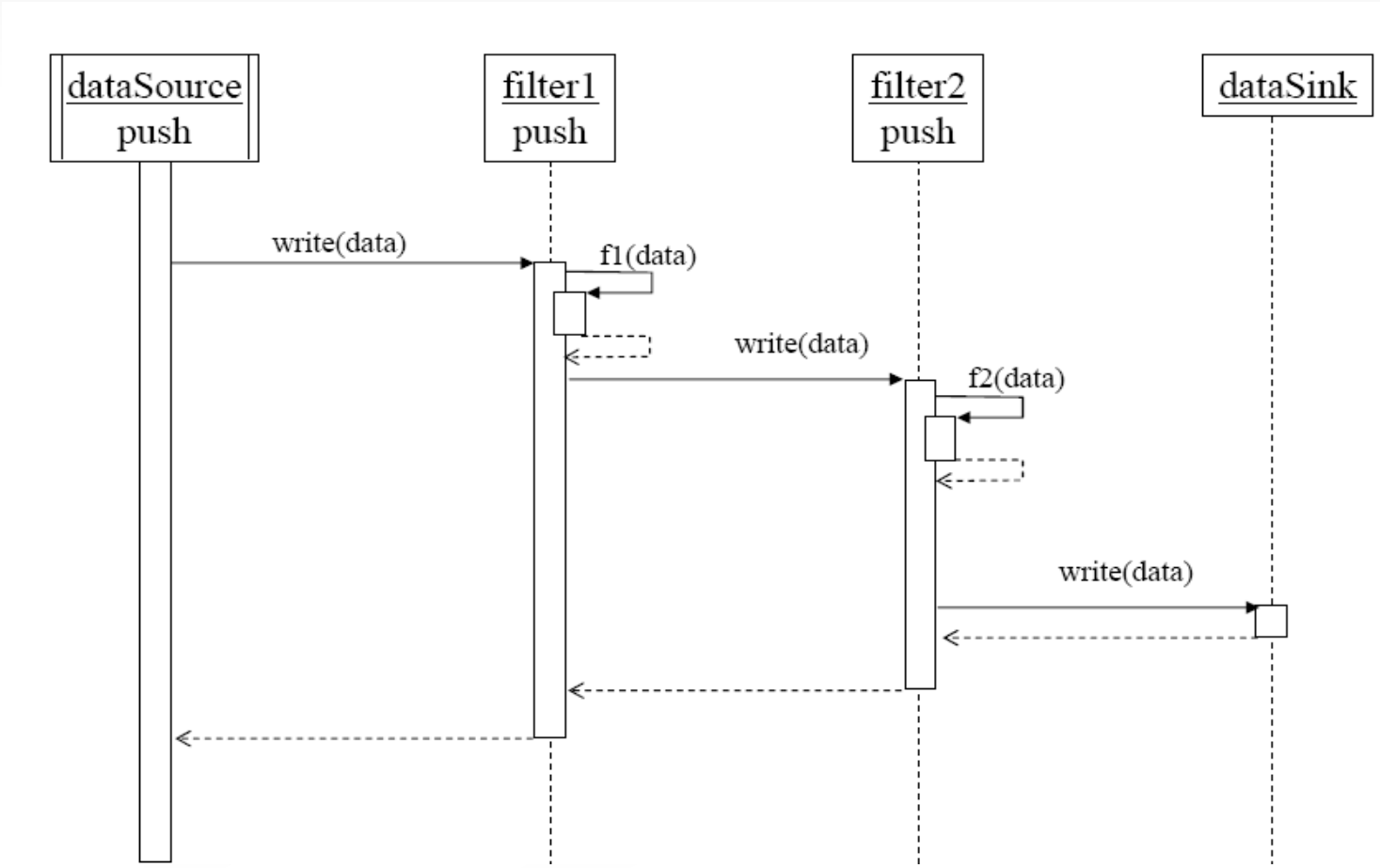
- Варианти
  - Филтрите може да са независими, без общо състояние и памет.
  - Филтърът не знае кой е преди и кой е след него в потока.
  - Коректността на резултата може да не зависи от реда, в който филтрите се изпълняват
- Къде ?
  - Unix pipes
  - lex/yacc-based compiler
    - Scanning->parsing->semantic analysis->code generation
  - Обработка на изображения
  - Сигнални обработки
  - Аудио и видео стриминг (audio and video streaming)

Pipes and Filters

- Основен проблем – ‘Какво кара данните да се движат’?
  - Зареждане (Push): източникът ги зарежда в следващия в потока
  - Извличане (Pull): приемникът (data sink) ги извлича от предходния
  - Комбинирана форма (Push/Pull): филтърът активно извлича данните от потока, извършва изчисленията и ги зарежда в потока на следващия.
    - Ако има повече от един такъв филтър е необходим механизъм за синхронизация.
  - Пасивно форма: нищи не извърва, а е само генератор на данни или приекник (sink).

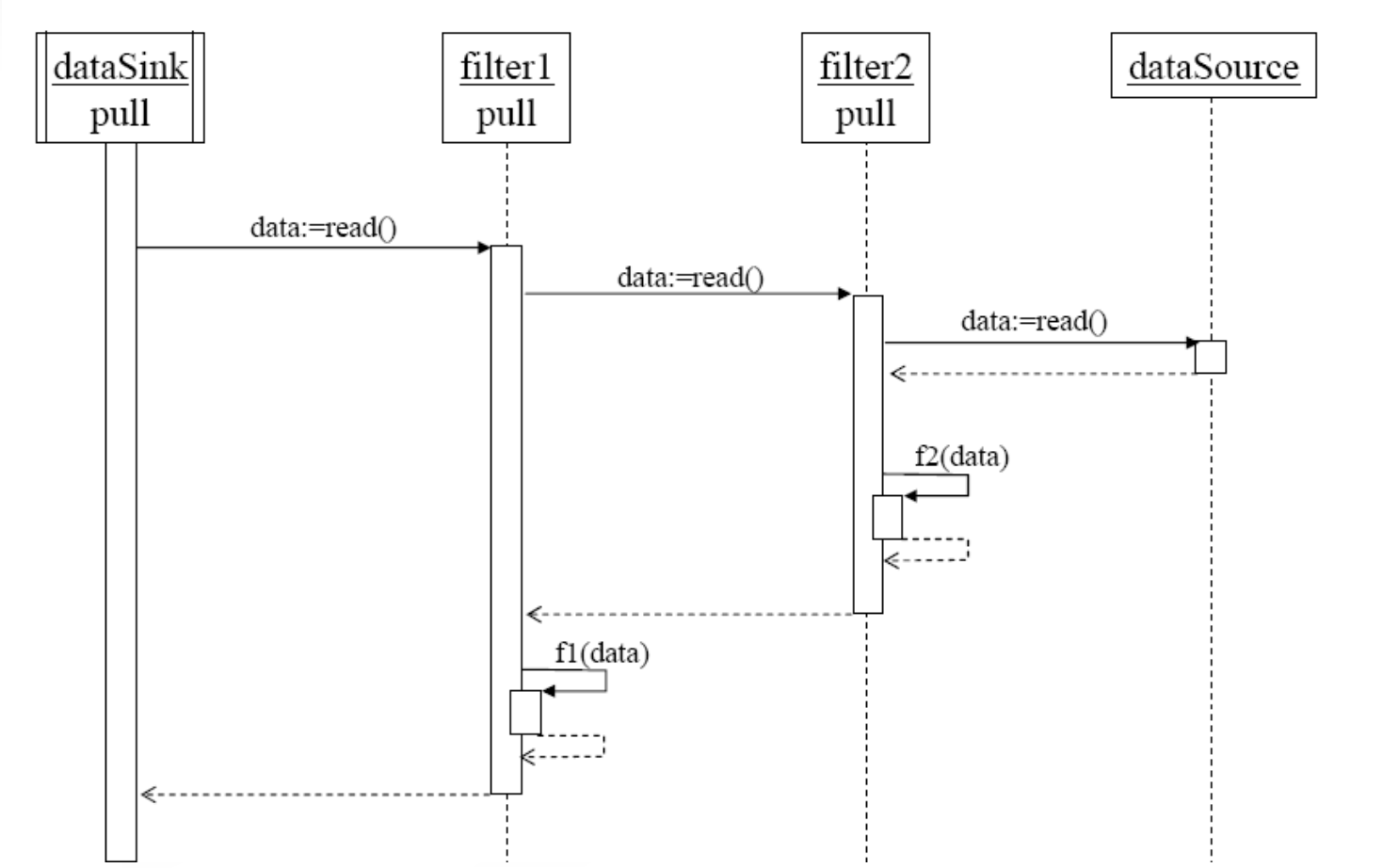
Зареждане и извличане на данни

- Зареждане (Push) с един активен източник



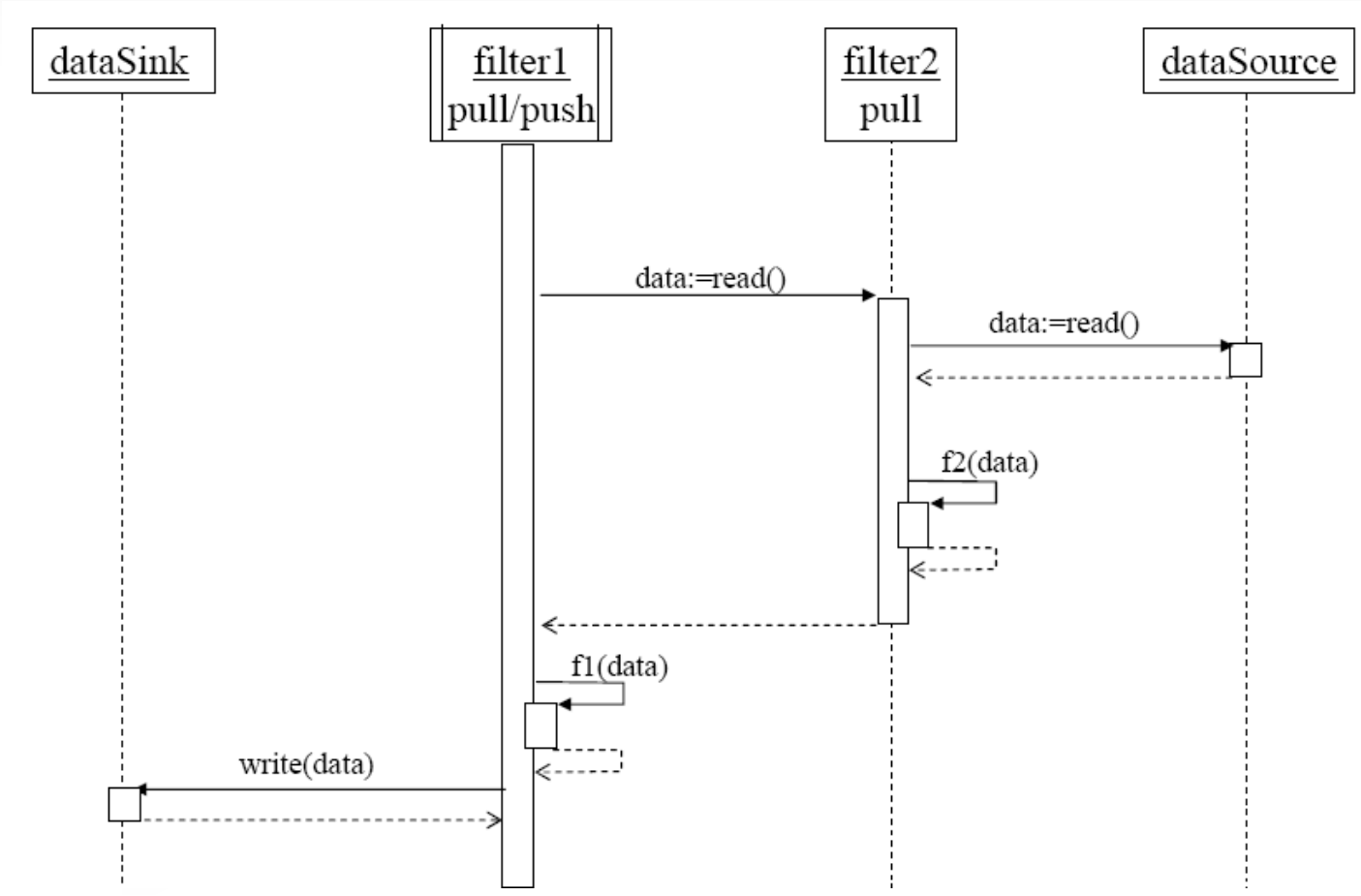
Зареждане и извличане на данни

- Извличане (Pull) с един активен приемник



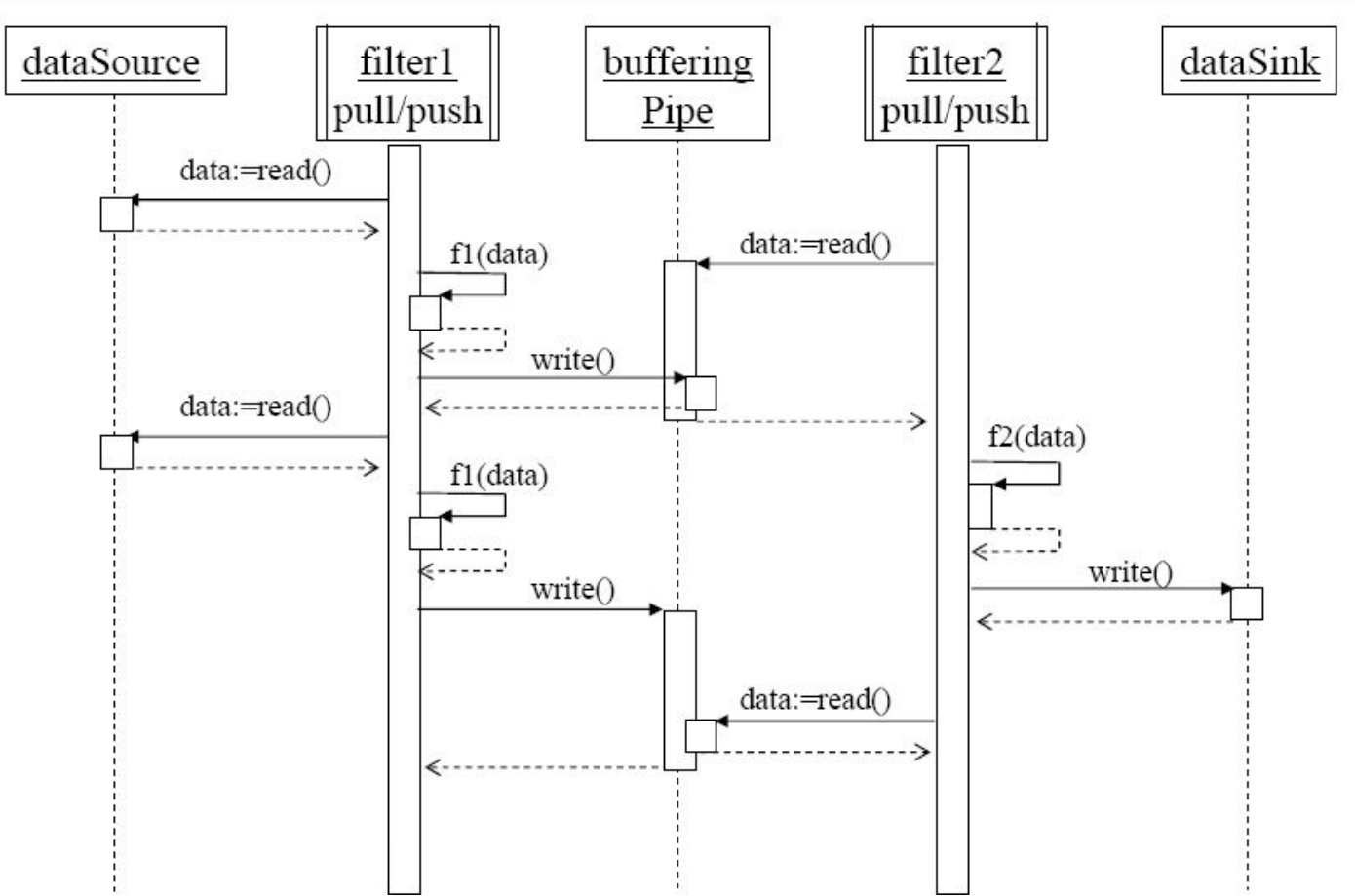
Зареждане и извличане на данни

- Комбинирана форма (Push-Pull) с пасивни източник и приемник



Зареждане и извличане на данни

- Комбинирана форма (Push-Pull) с активни филтри и синхронизационен буфер



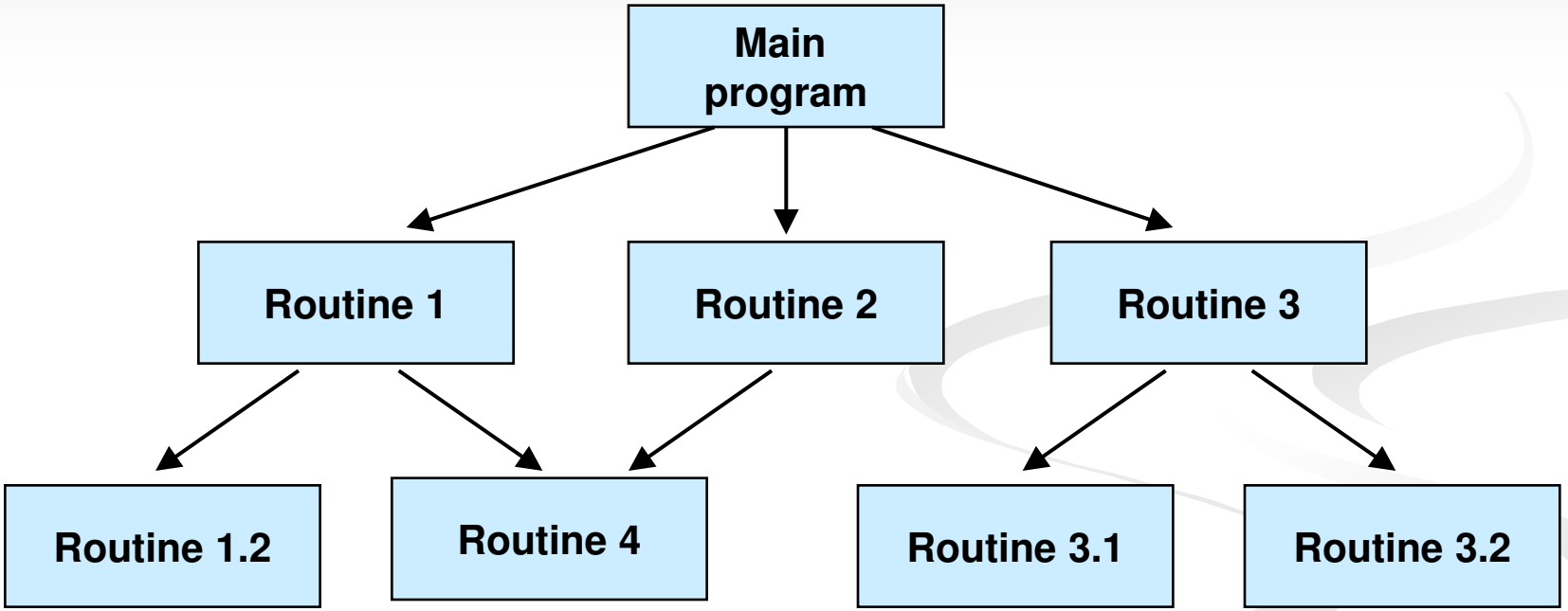
Pipes and Filters

- Проблеми за изследване
  - ✓ Презиползваемост – формата на данните за пренос
  - ✓ Заменяемост
  - ✓ Формален анализ, включително определяне на взаимна блокировка (deadlock detection).
  - ✓ Паралелност на изпълнението на филтрите.
  - ✓ Синхронизацията на потоците може да промени архитектурата.
  - ✓ Error handling
  - ✓ Изборът на типа на данните за потока.

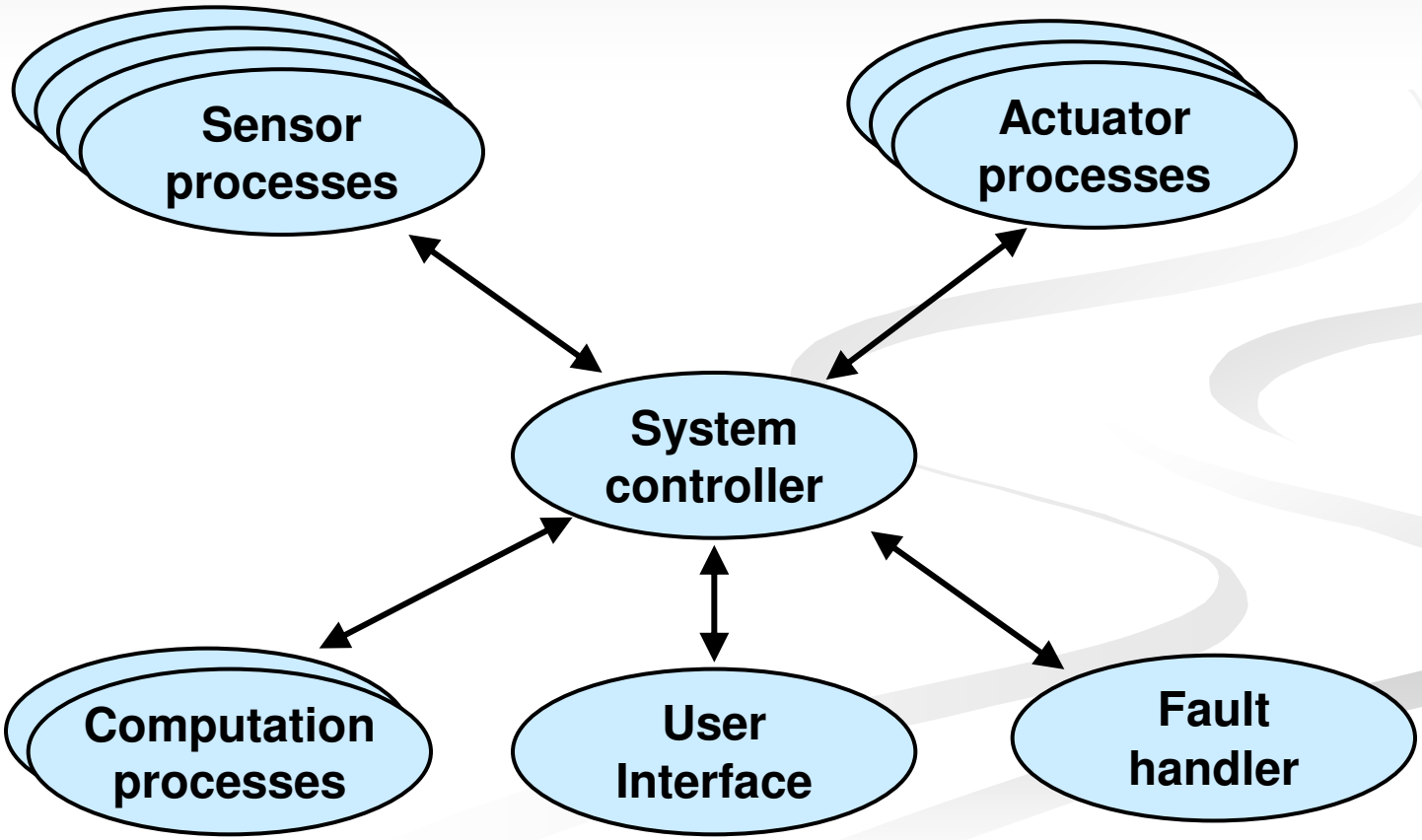
Архитектура с централизирано управление

- Главна програма + архитектура, базирана на подпрограми
  - ✓ Йерархична декомпозиция
  - ✓ Данните се предават като параметри.
  - ✓ Главната програма управлява последователността от обработки през подпрограмите

Call-Return Model



Management Control



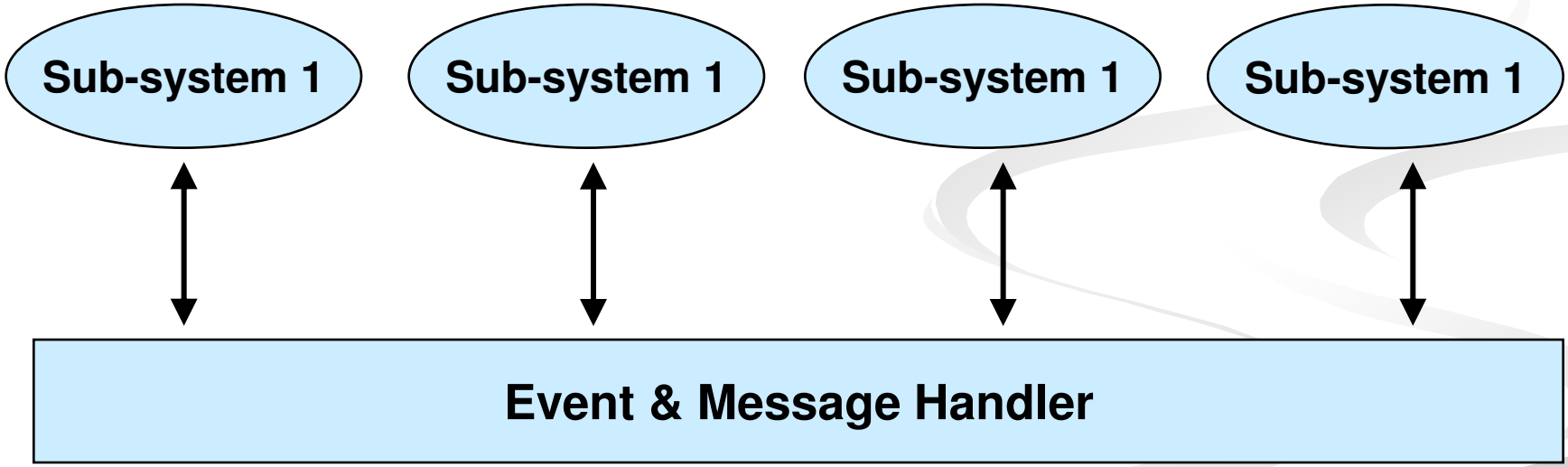
ОО архитектура

- Без наследственост има архитектурата е от друг вид.
- Основни проблеми за проверка:
  - ✓ Правилно разделяне на проблемите между свързаните агенти.
  - ✓ Промяната на имплементацията отразява ли се на клиентите ?
    - × Ако B се промени то това ще доведе ли до промяна на изпозлващите го обекти.
    - × A използва B и ако C използва B, то ако промяната на C се отрази върху B то дали има неочаквана промяна на A.
  - ✓ Разпределена функционалност като резултат от сложни динамични връзки.

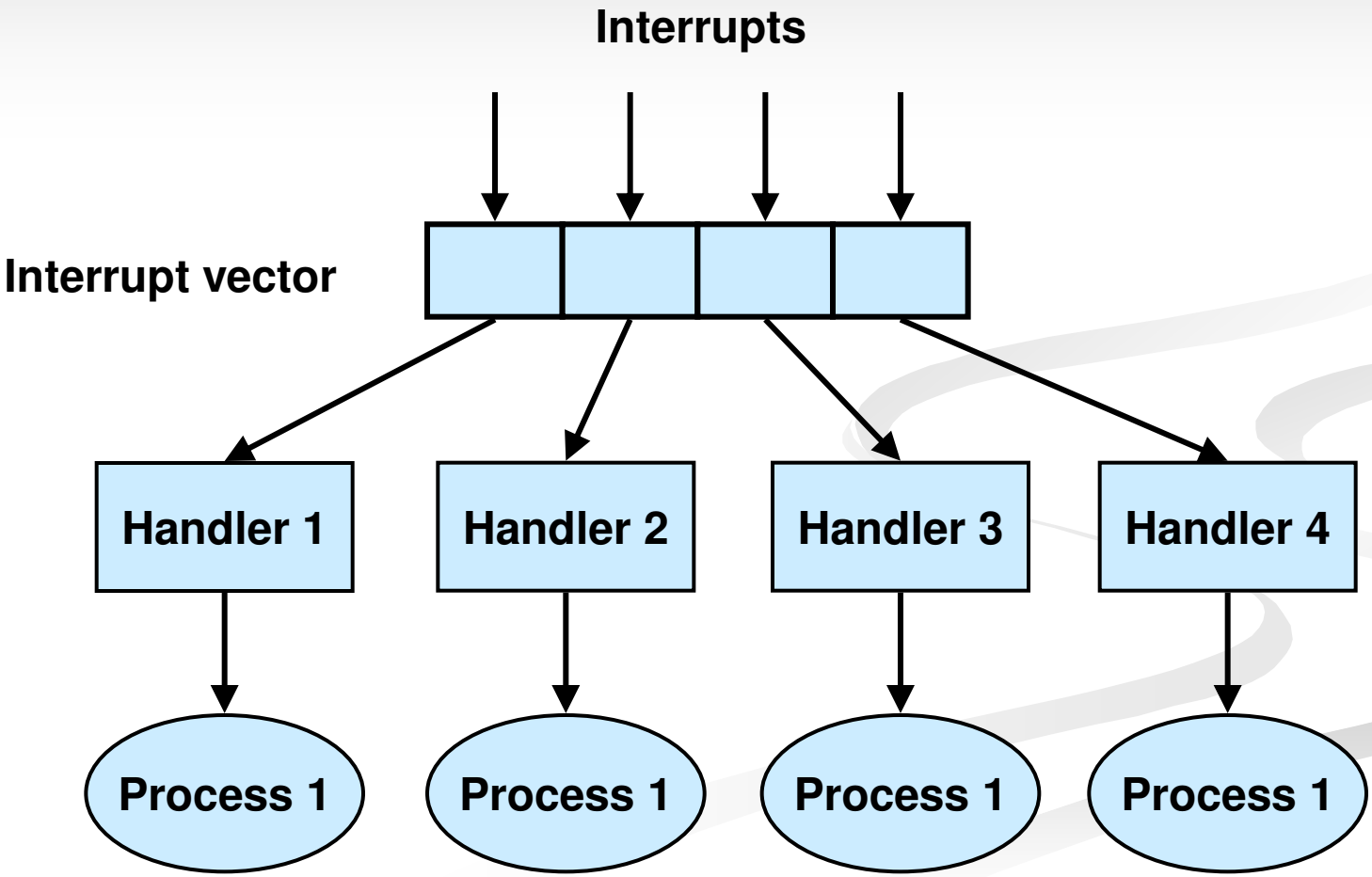
Event-Based

- Тази архитектура се характеризира със стила на комуникация между обектите:
  - ✓ Директна активизация (procedure invocation), изпращане на съобщение (message) до broadcast или multicast (до регистрирани компоненти), събития (events).
  - ✓ Компонентите не са задължително обекти.
- Видове
  - ✓ С разпространение (broadcast)
  - ✓ Системи с прекъсване (interrupt-driven)

Broadcasting



Interrupt-Driven Control



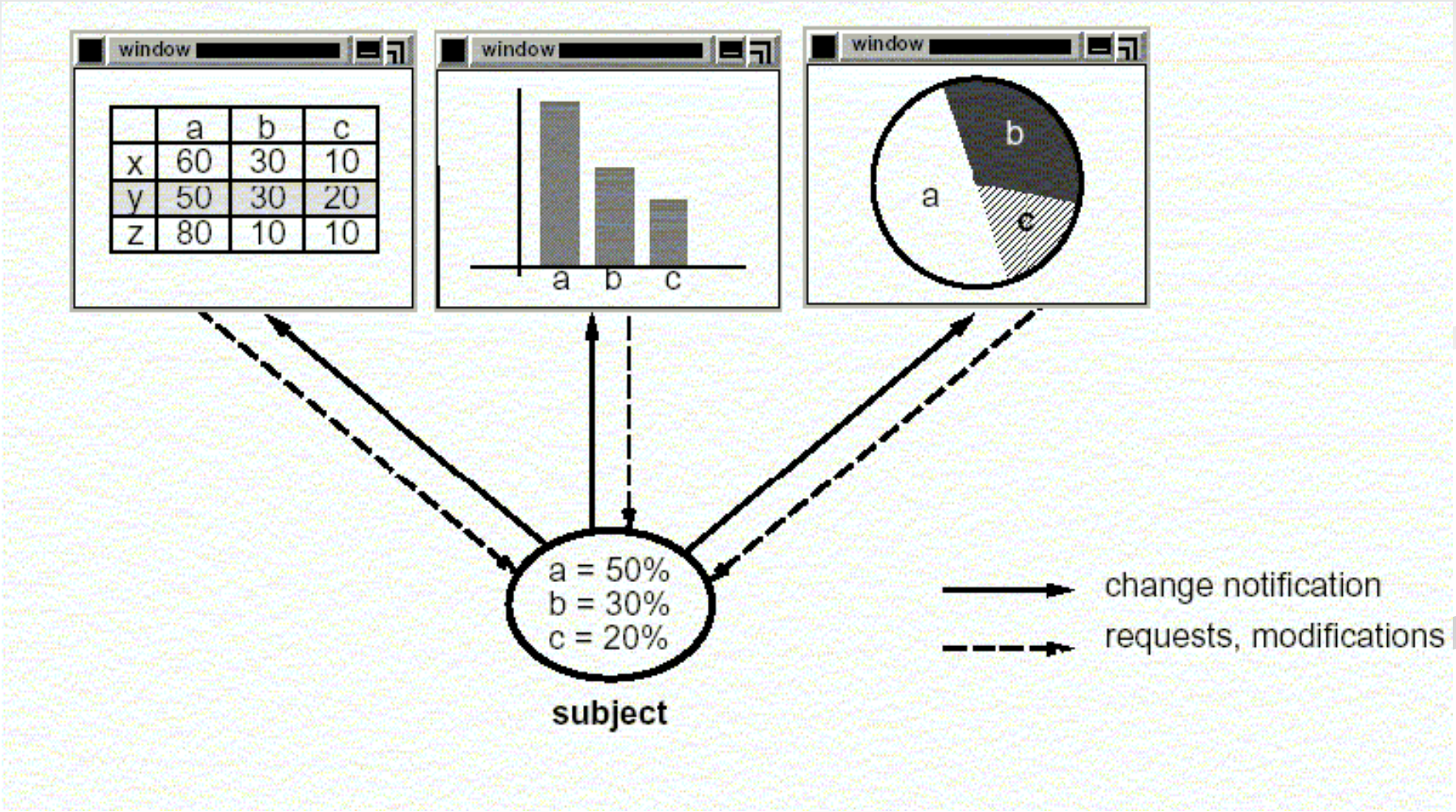
Явно активизиране

- Основни проблеми
  - Загуба на контрол
    - При генерирано събитие компонентът не знае дали някой го възприема.
    - Не знае дали някой ще бъде активизиран.
    - Не знае кога свършва работата на активизирания.
  - Коректността зависи от контекства, при който е възникнала активизацията, т.е. Непредсказуеми взаимодействия.
  - Общи данни – имплицитното и експлицитна активизация се използват паралелно.

Model-View-Controller

- Декомпозиция на интерактивна система на три компонента:
  - ✓ Model – основната функционалност и данни
  - ✓ Едно или повече ‘Views’ – визуализират информацията за потребителя
  - ✓ Един или повече ‘Controllers’ – управляват входа от потребителя.
- Основен проблем
  - ✓ Механизъм за защитена промяна (change-propagation mechanism) – да се осигурява консистентност между потребителския интерфейс и модела
  - ✓ Промяната на модела пред един контролер, свързан с даден изглед (‘view’) трябва автоматично да обнови другите изгледи.
  - ✓ Съществува ли обвързаност между контекста на контролера и изгледа – може да доведе до реализирането им като един компонент.

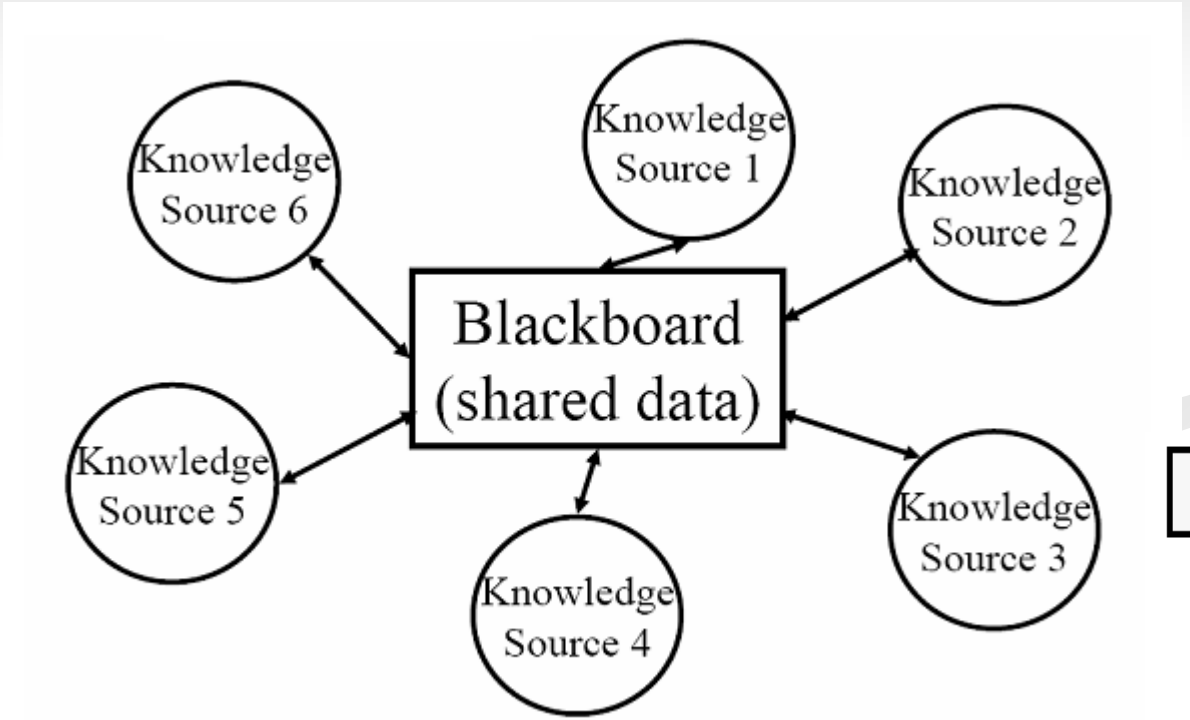
Model-View-Controller

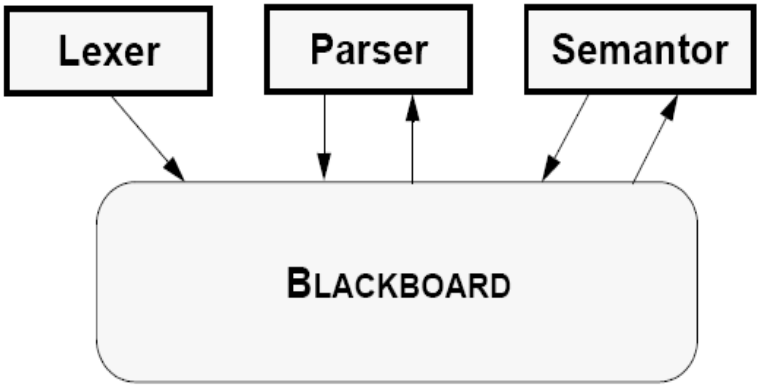


Централизирано съхранение на данните (Repositories / Data Centred)

- Един централен компонент, съхраняващ централизирано данните и представящ състоянието на системата, и множество независими компоненти, работещи върху съхраненяваните данни.
- Проблем – връзката между външните компоненти и хранилището:
  - ✓ *Транзакционни БД (Transactional databases)*:
    - × Входният поток от транзакции управлява процесите за управляние на данните в хранилището.
    - × Пасивна идеология.
  - ✓ *‘Черна дъска’ (Blackboard architecture)*:
    - × Текущото състояние на данните в хранилището управлява процесите.
    - × Активна идеология.

Централизирано съхранение на данните





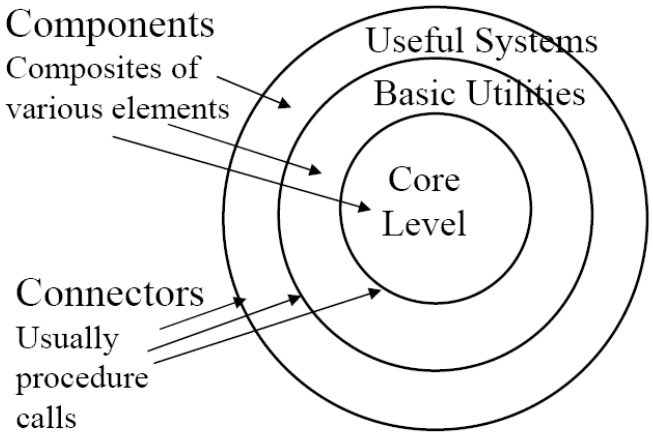
Централизирано съхранение на данните

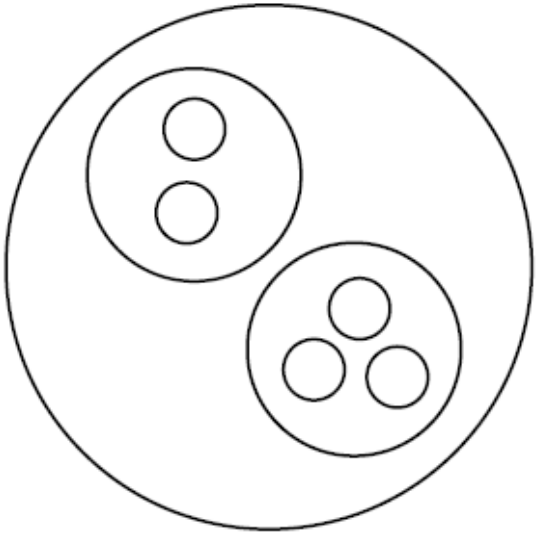
- Примери:
  - ✓ Информационните системи
  - ✓ Интегрирани среди за разработка
  - ✓ Графични редактори
  - ✓ Системи за ИИ
  - ✓ Други.
- Проблеми:
  - ✓ Голямо количество на данните
  - ✓ Централизирано управление на данните.
  - ✓ Разпределени данни
  - ✓ Еволюция

Йерархична архитектура

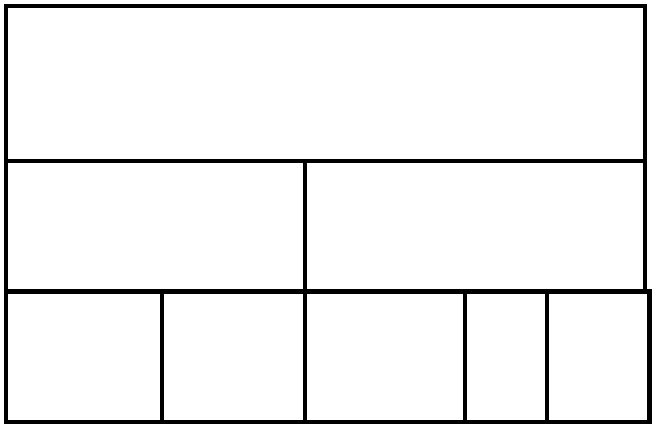
- Проблеми
  - ✓ Системна организация
    - × Многонивова ‘Клиент-Сървър’
    - × Всяко ниво експортира “API” за горното ниво
  - ✓ Всяко ниво се явява едновременно
    - × *Сървър* – за горното ниво
    - × *Клиент* – за долното ниво
  - ✓ ‘Чисто’ йерархична система ли е ?
  - ✓ Вид:
    - × Интерпретативна или
    - × Многонивова

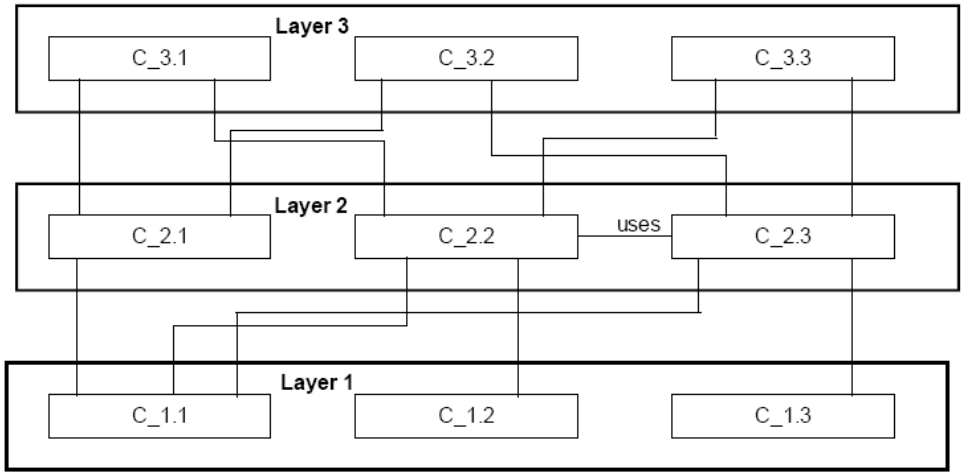
Йерархична архитектура





Йерархична архитектура





Йерархична архитектура

- Стратегии за ‘error handling’
  - ✓ Има или няма пропагация на грешката!
  - ✓ Количество на грешките като тип и брой при пропагация през нивата
    - × Проблем с обединяването на няколко грешки в една на горното ниво
    - × Проблеми с контекста на грешките при пропагация
- Как промяната на реализацията на дадено ниво се отразява на свързаните с него ?
- Как се управлява абстракцията ?
- Как се управлява бързодействието и сигурността ?

Архитектура ‘Process-Control’

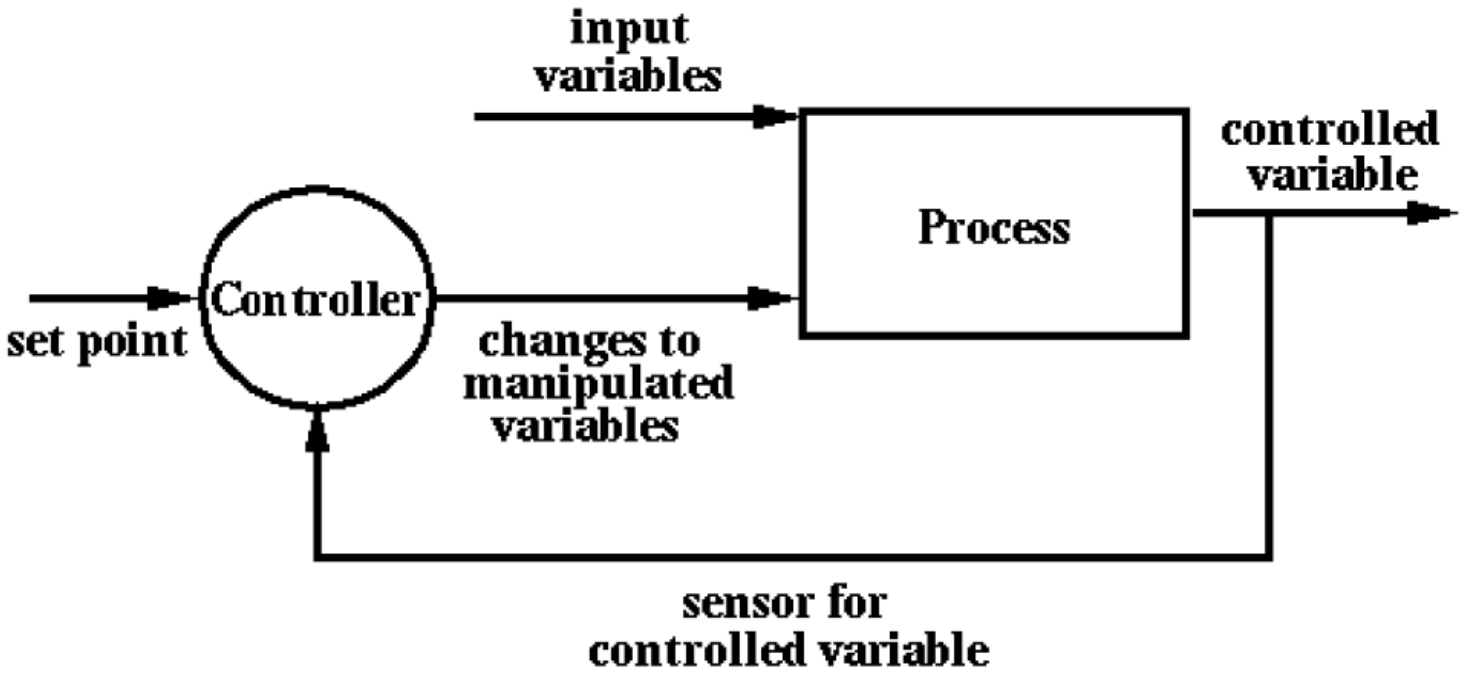
- Подходящ за приложения, чиято цел е да поддържа определени свойства на изходите на процеса на достатъчно близо до референтни стойности.
- Компоненти
  - ✓ Процеси (‘Process’) – механизми за манипулация на една или повече процесни променливи
  - ✓ Алгоритъм за управление – управлява процеса на манипулация на процесните променливи.

Архитектура ‘Process-Control’

- Конектори – това са взаимовръзките в потока на данни
  - Процесни променливи
    - Контролиран променлива –предназначена е да се контролира от системата.
    - Манипулирани променлива - стойността им може да бъде променена от управлението. от администратора.
    - Входната променлива (set point) – това е желаната стойност за манипулирана променлива.
  - Сензори

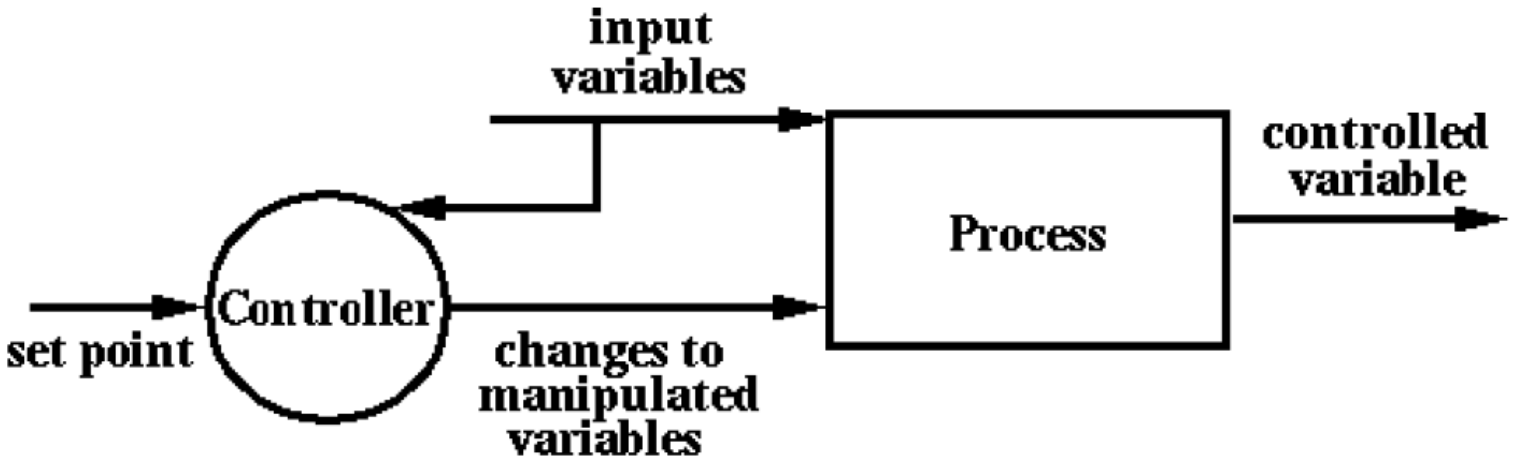
Архитектура ‘Process-Control’

- Системи с обратна връзка



Архитектура ‘Process-Control’

- Системи с отворен контур



Questions ?

Системи за реално време

Видове програмни модули (Програмиране от високо ниво)

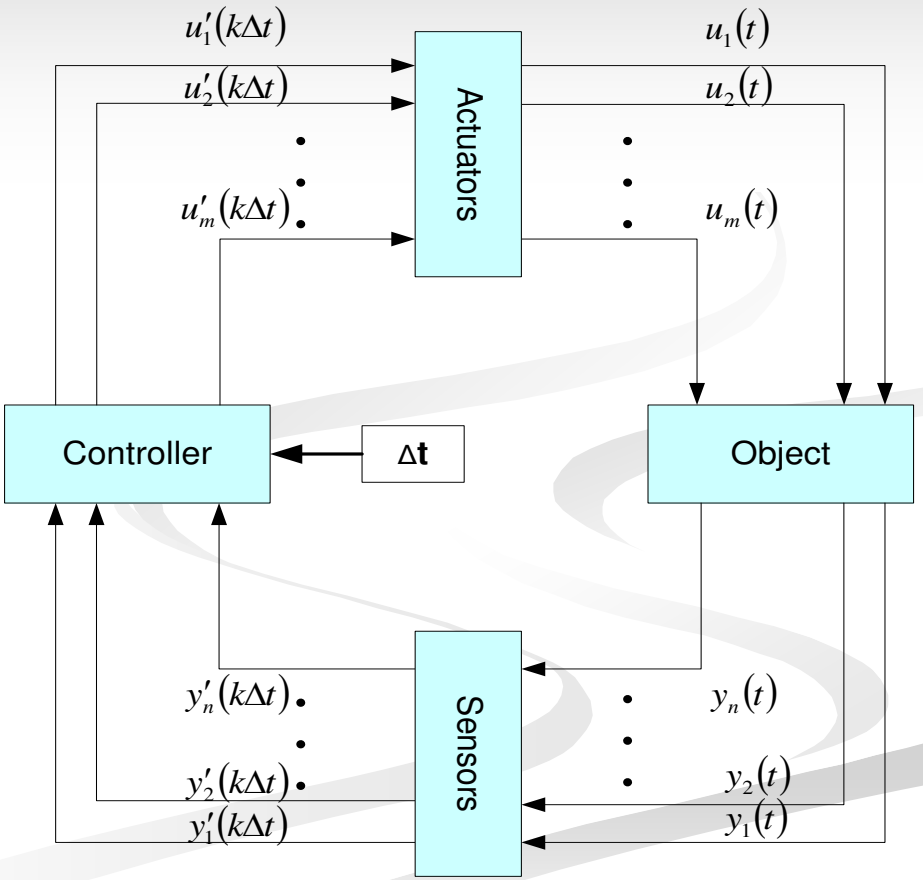
- Операционна среда
- Ядро
- Задачи (tasks)
- Задачи за обслужване на прекъсванията (ISR-tasks)

Базова структура на СРВ

Управляваща част

Управлявана част

Операционна среда



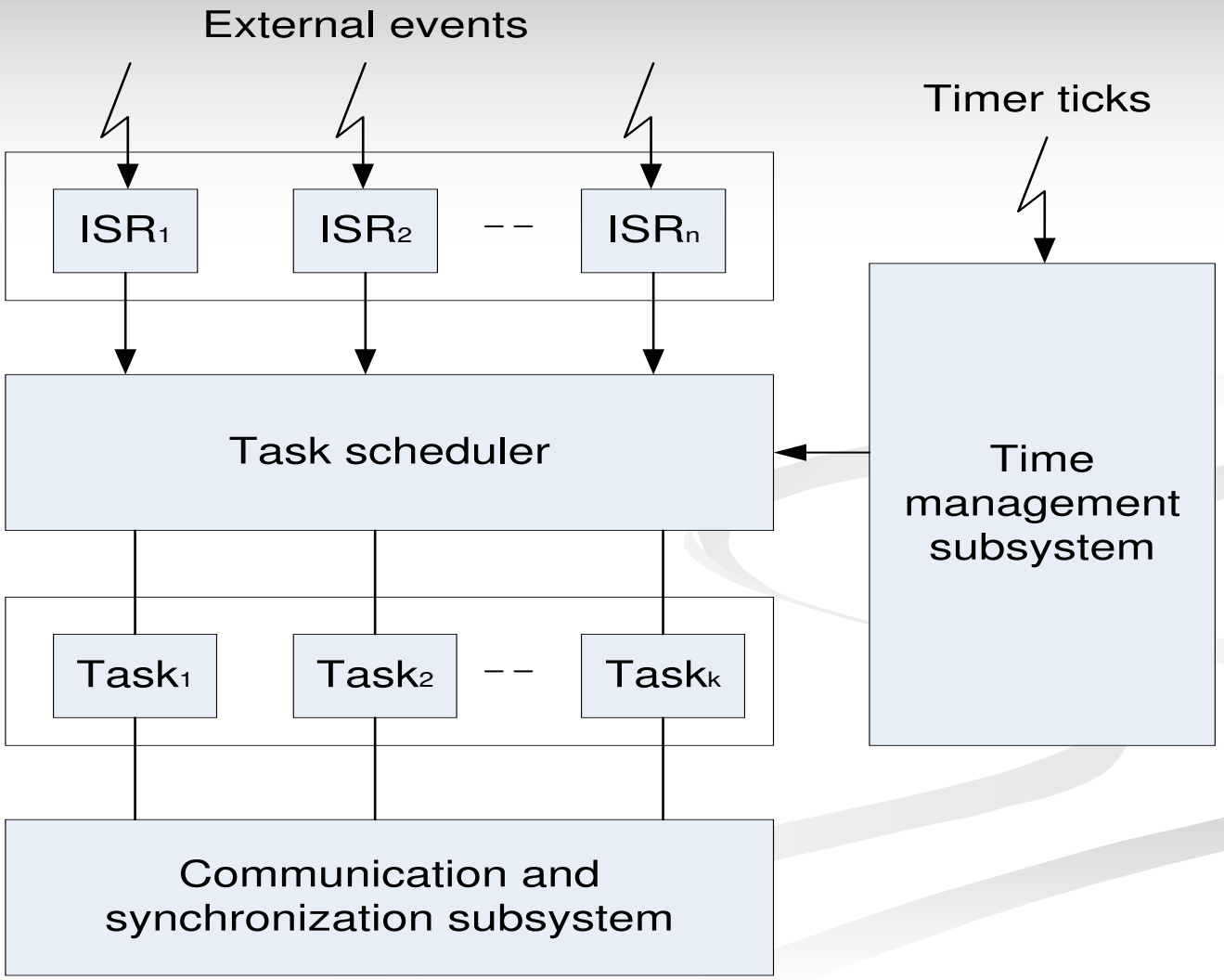
Операционна система за реално време (ОСРВ)

- Дефиниция:
  - ✓ *“Една компютърна система е система за реално време когато коректността на резултатите зависи не само от логиката, поради която са получени, но и от момента на тяхното създаване.”*
  - ✓ *Коректен изход = Коректен резултат + Коректно време*

Характеристики на ОСРВ

- Главна задача на диспечера на ОСРВ - да следи за крайния срок (deadline).
- Система трябва да реагира своевременно и по предсказуем начин на външни стимули, появяващи се в непредвидим момент.
  - ✓ Тя трябва да има предсказуемо поведение при всички възможни сценарии.
  - ✓ Тя е само един модул и сама не може да гарантира коректността на цялата система.
- ОСРВ не е транзакционна система.
- Главна задача на диспечера на ОСРВ:
  - ✓ Да следи за крайния срок (deadline)

Структура на ОСРВ



Видове ОСРВ

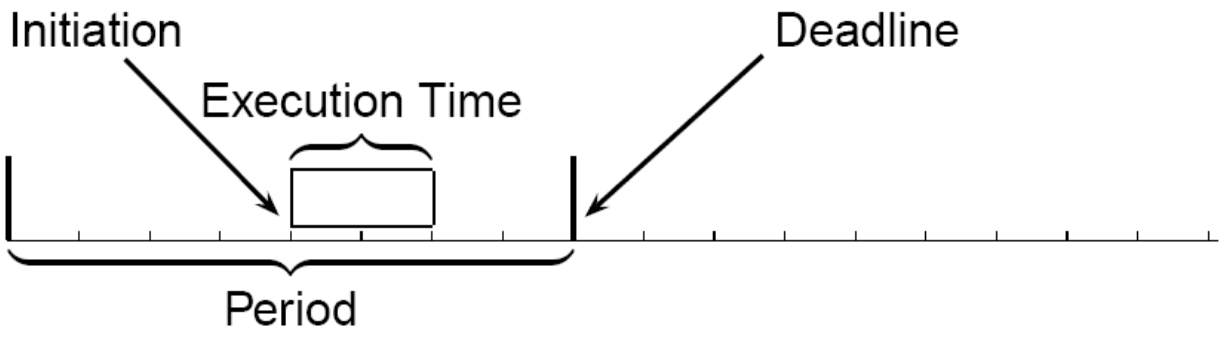
- С твърди времеви ограничения (Hard Real-Time)
  - ✓ Неспазването на крайните интервали има катастрофален резултата за системата.
- Firm Real-Time
  - ✓ Неспазването на крайните интервали води до непредсказуема редукция на качеството
- С меки времеви ограничения (Soft Real-Time)
  - ✓ Крайните срокове могат да се пропускат или могат да се възстановяват
  - ✓ Редукцията на качеството е приемлива.
- Без времеви ограничения (Non Real-Time)
  - ✓ Няма крайни срокове.

Ограничения, влияещи върху задачите в една ОСРВ

- Ограничения от крайни срокове (Deadlines)
- Ресурсни ограничения
  - ✓ За да се постигне по-висока надеждност за изпълнение на задачата
  - ✓ Необходим е излишък за правилно функциониране.
- Приоритетни ограничения
  - ✓ T_1 -> T_2: Задачата T_2 само след като T_1 завършила изпълнението си.
- Отказоустойчивост
  - ✓ За да се постигне по-висока надеждност за изпълнение на задачата.
  - ✓ Необходим е излишък за правилно функциониране.

Видове реално-времеви задачи

- Според ограничение по време:
  - ✓ Твърди (Hard real-time) задачи
  - ✓ Меки (Soft real-time) задачи
- Според момент на активация
  - ✓ Периодични
  - ✓ Апериодични
  - ✓ Спорадични



Реално-времеви задачи

	Hard Real-Time	Soft Real-Time
Време за реакция	твърди граници	меки граници
При върхово натоварване	предсказуемо	деградиране
Управление на пространството	от средата	от компютъра
Надеждност	критична	не-критична
Излишък	активен	възстановим
Интегритет на данните	кратковременен	дълговременен
Откриване на грешки	автоматичен	външно управляемо

Реално-времеви задачи

- Периодични
  - ✓ Управлявани от време (Time-driven).
  - ✓ Характеристиките *са априорно* зададени
    - Задачата T_i се характеризира чрез периода си, най-лошо време за изпълнение и краен период
- Апериодични
  - ✓ Управлявани от събития (Event-driven).
  - ✓ Характеристиките *не са априорно* зададени
    - Задачата T_i се характеризира чрез време за получаване, време за изпълнение, най-лошо време за изпълнение и краен период.
- Спорадични – апериодични, за които се знае минималното време за появяване

Ядра за ОСРВ

- Монолитни
  - ✓ Един общ кодов фрагмент
  - ✓ Малък размер и проста вътрешна структура
  - ✓ Трудни за промяна и настройка
- Многослойни
  - ✓ Разделени на подсистеми – най-често се използва ОО дизайн
  - ✓ По-добра изолация от потребителските задачи
  - ✓ По сложни, с по-голямо време на реакция, но стабилни
- Разпределени
  - ✓ Разпределената система се представя като *virtual unicomputer*

Диспечиране на задачите

- Основна цел
  - ✓ Управление на времето на процесора
- Възможни елементи за управление:
  - ✓ Процеси (Processes)
  - ✓ Нишки (Threads)
  - ✓ Фибри(Fibers)

Те са леки (lightweight) процеси.

Много леки елемент, управлявани от приложението.

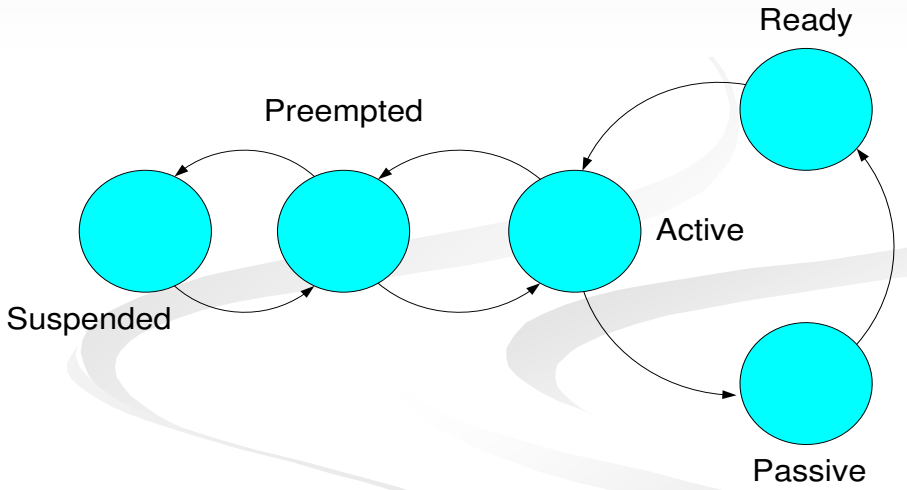
Състоят се от:
- Само код.

Не се използват в повечето ОСРВ.

За какво: Предварителен дизайн и симулация на реално-времеви нишки.

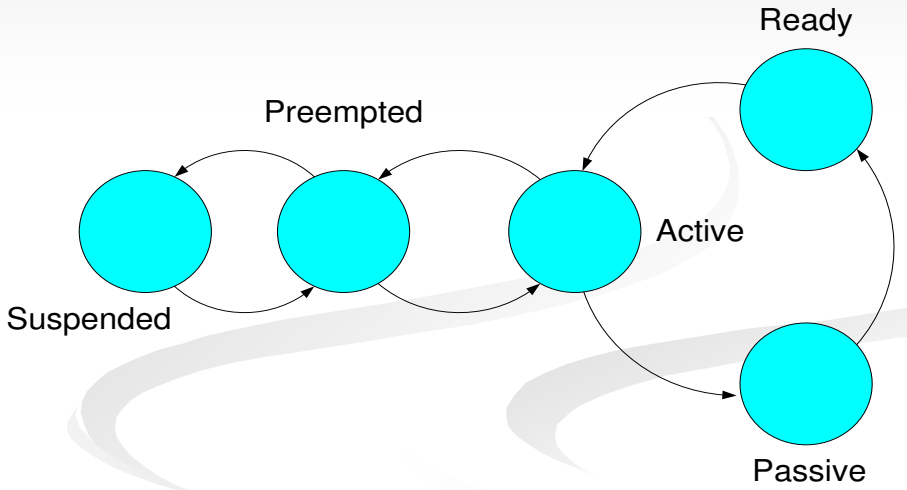
Граф на състоянието на задачите

- Активна задача (active)
  - ✓ Тя работи с процесора
- Пасивна задача (passive)
  - ✓ Съществува, но не е избрана за изпълнение
- Прекратена задача (suspended)
  - ✓ Била е стартирана, но не може да се изпълнява по някаква причина
- Изместена задача (preempted)
  - ✓ Била е стартирана, е изместена по някаква причина
- Готова задача (ready)
  - ✓ Изчаква да бъде активизирана.



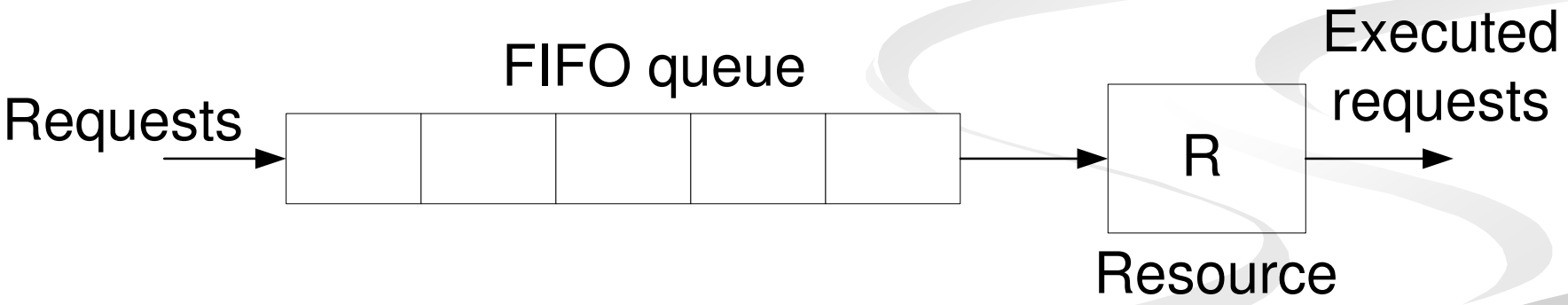
Промяна на състоянието на задачите

- Active -> Passive
  - ✓ Задачата завършва (експлицитно или поради скрито системен извикване: ‘schedule’, ‘quit’, др.)
- Passive -> Ready
  - ✓ Системно API-извикване (‘execute’,‘invoke’, др.)
- Ready -> Active
  - ✓ по диспечерно решение
- Active -> Preempted
  - ✓ поява на диспечируемо събитие преди края на задачата
- Preempted -> Active
  - ✓ по диспечерно решение
- Active -> Suspended (via Preempted)
  - ✓ Най-често нещо необходимо е заето.
- Suspended -> Preempted
  - ✓ След освобождаване на нужен ресурс или подобно.



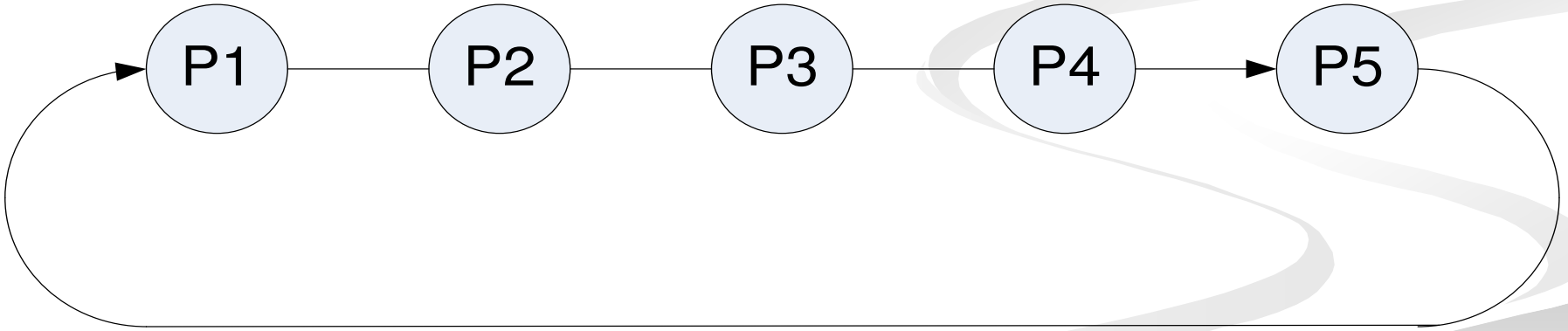
Управление на задачите

- Опашка (FIFO)



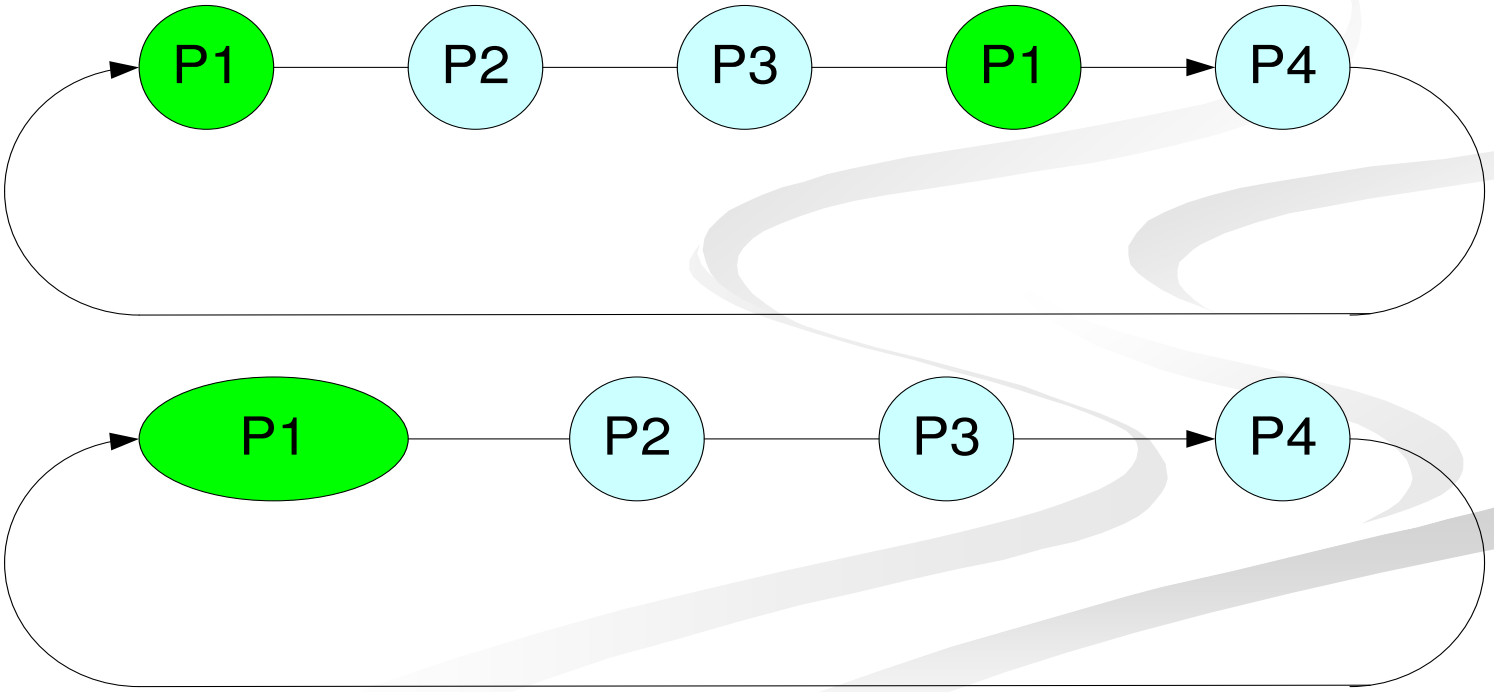
Управление на задачите

- Циклично заемане на процесора (Round-robin – Simple)



Управление на задачите

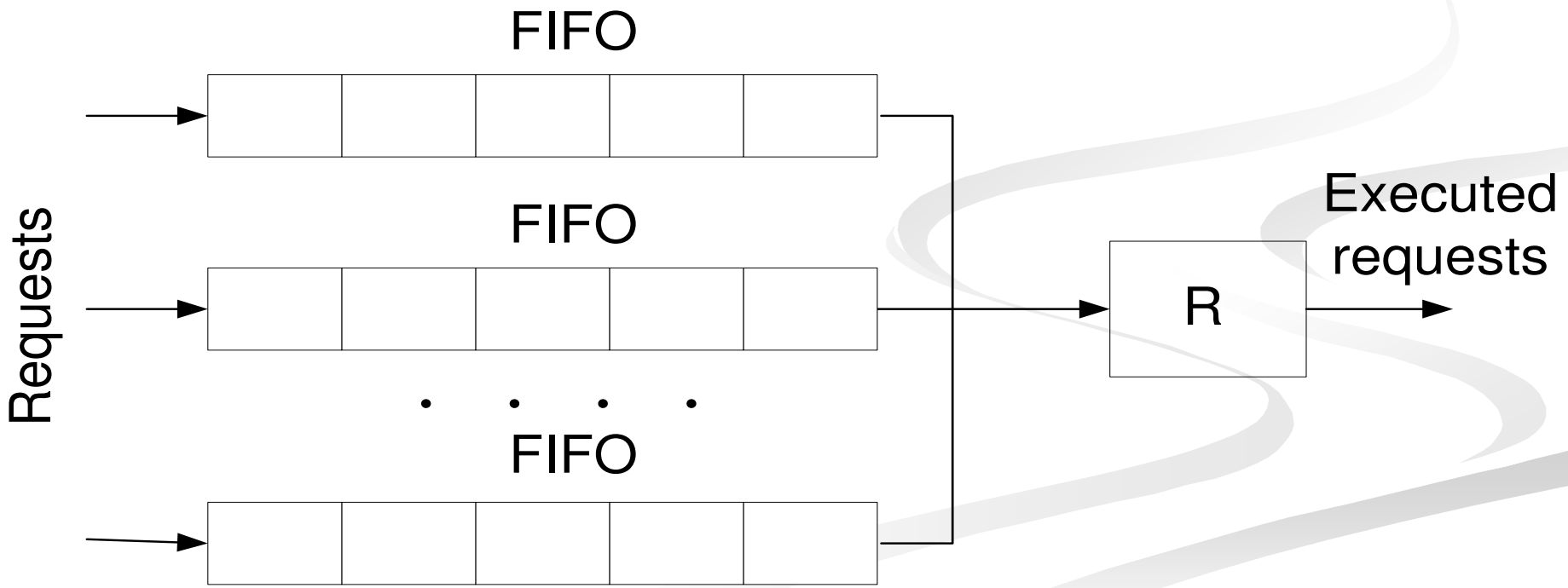
- Тегловно базирано циклично заемане на процесора (Round-robin – Weighted)



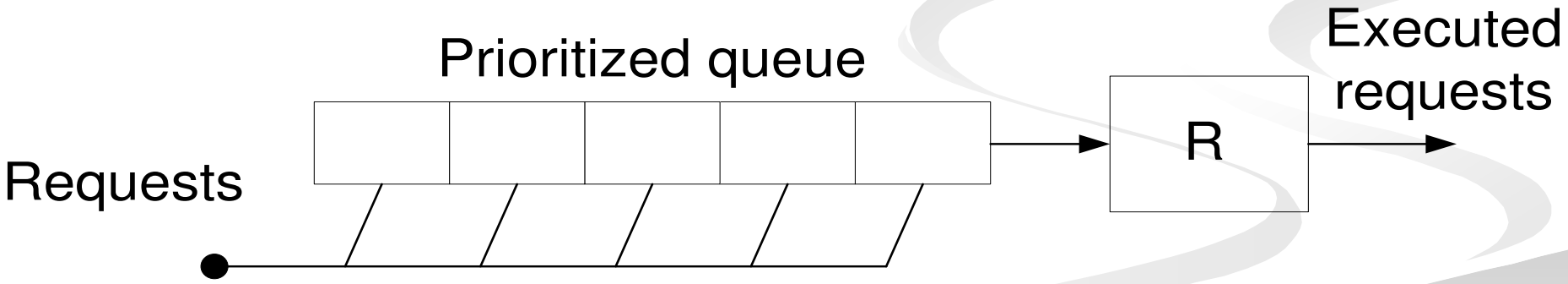
Управление на задачите

- Приоритетна диспечеризация
  - ✓ Приоритетни опашки (Priority queues)
  - ✓ Приоритизирани опашки (Prioritized queues)
  - ✓ Смесена дисциплина на диспечиране (cyclic priority scheme)
  - ✓ Динамична дисциплина на диспечиране

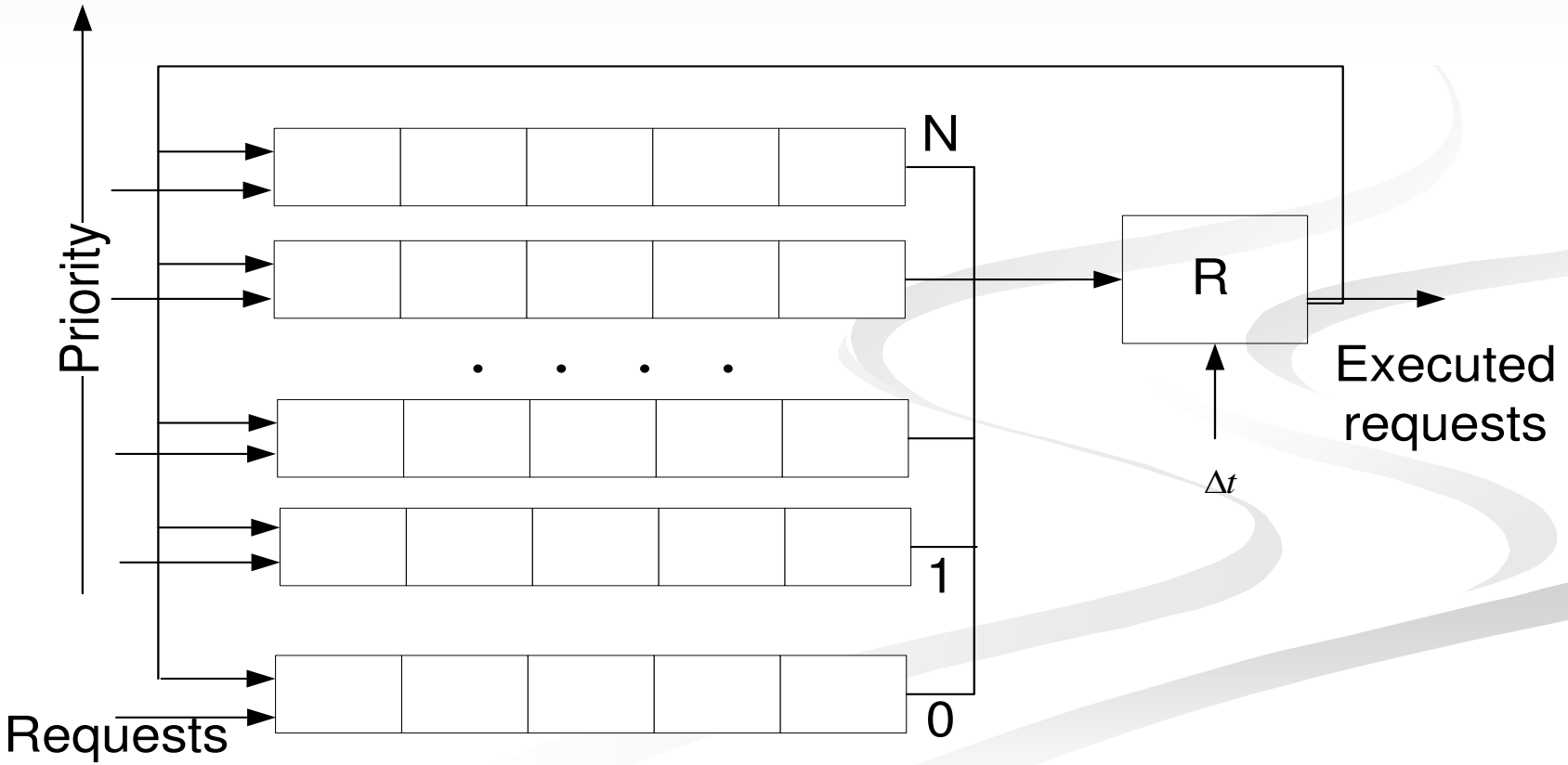
Приоритетни опашки



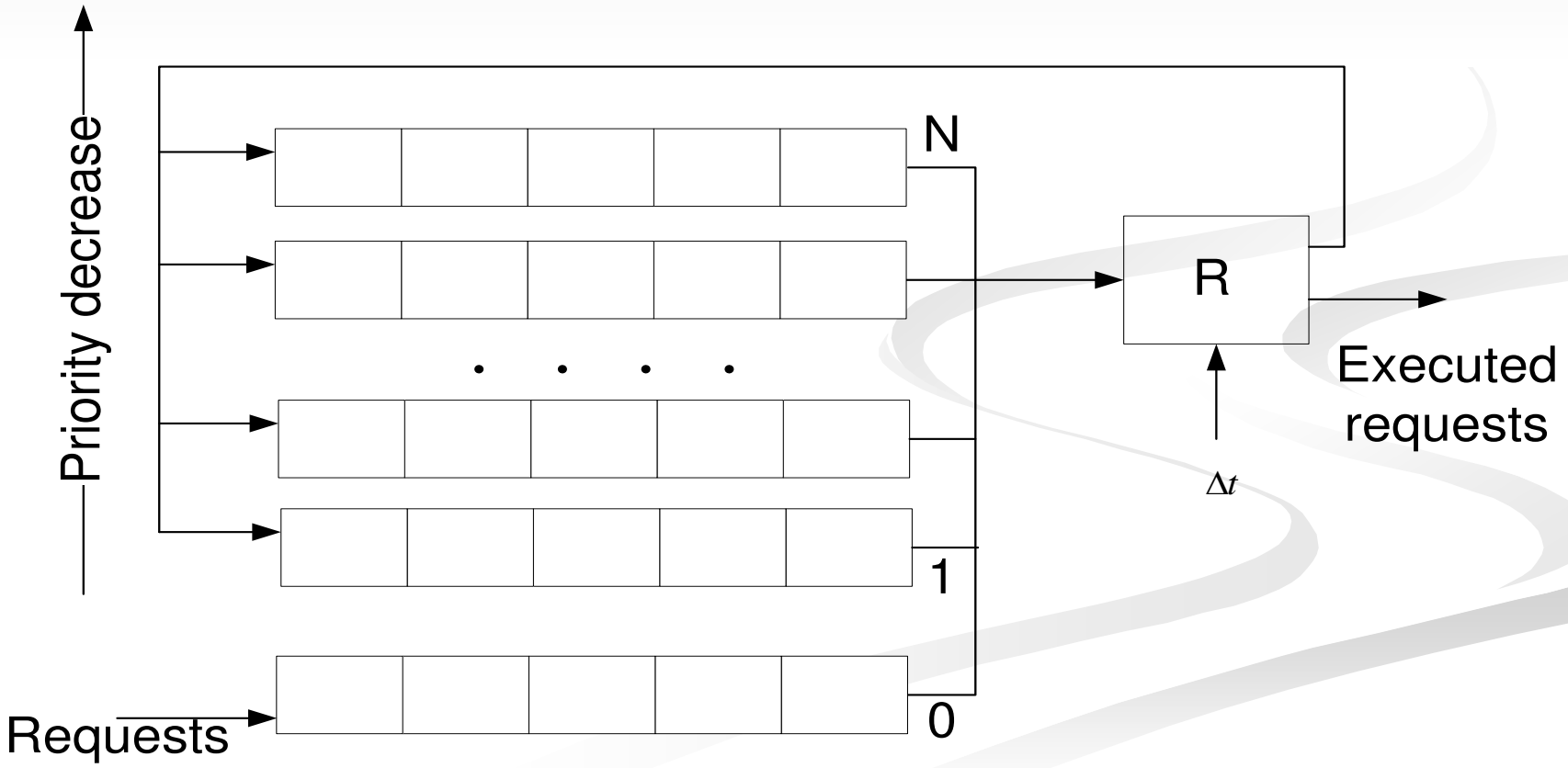
Приоритизирани опашки



Смесена дисциплина на диспечиране

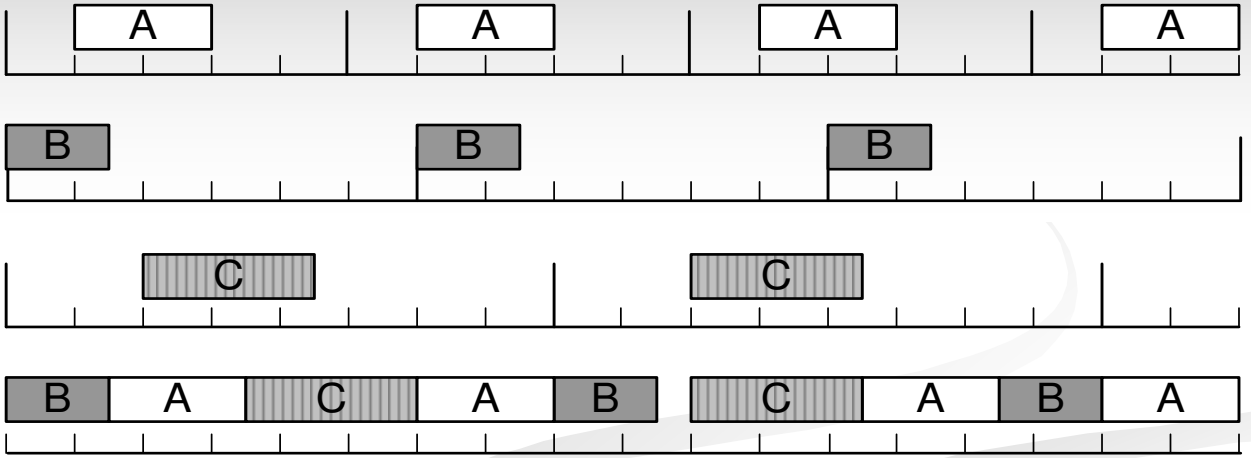


Динамична дисциплина на диспечиране

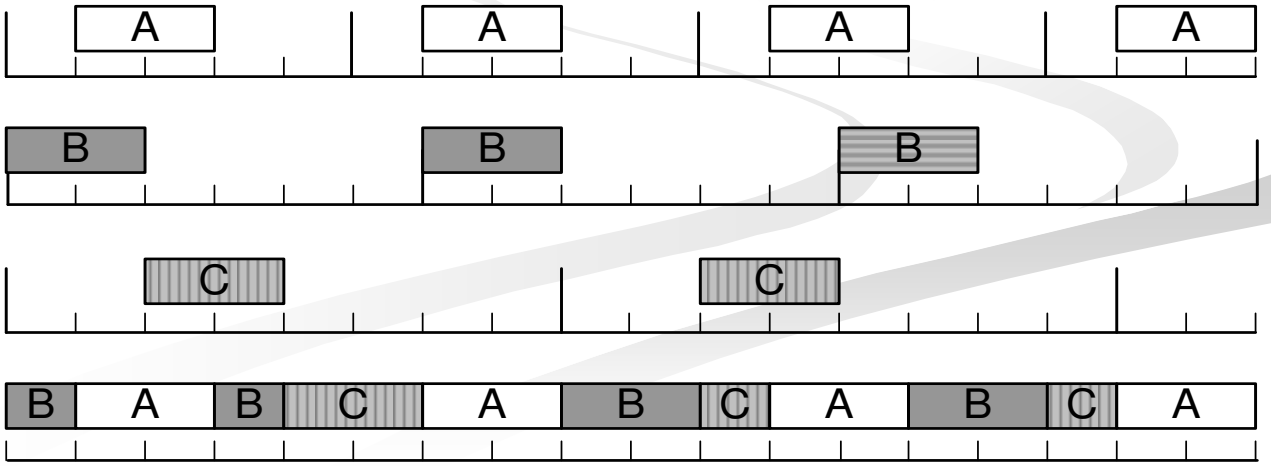


Дисциплина на диспечиране

Относителна



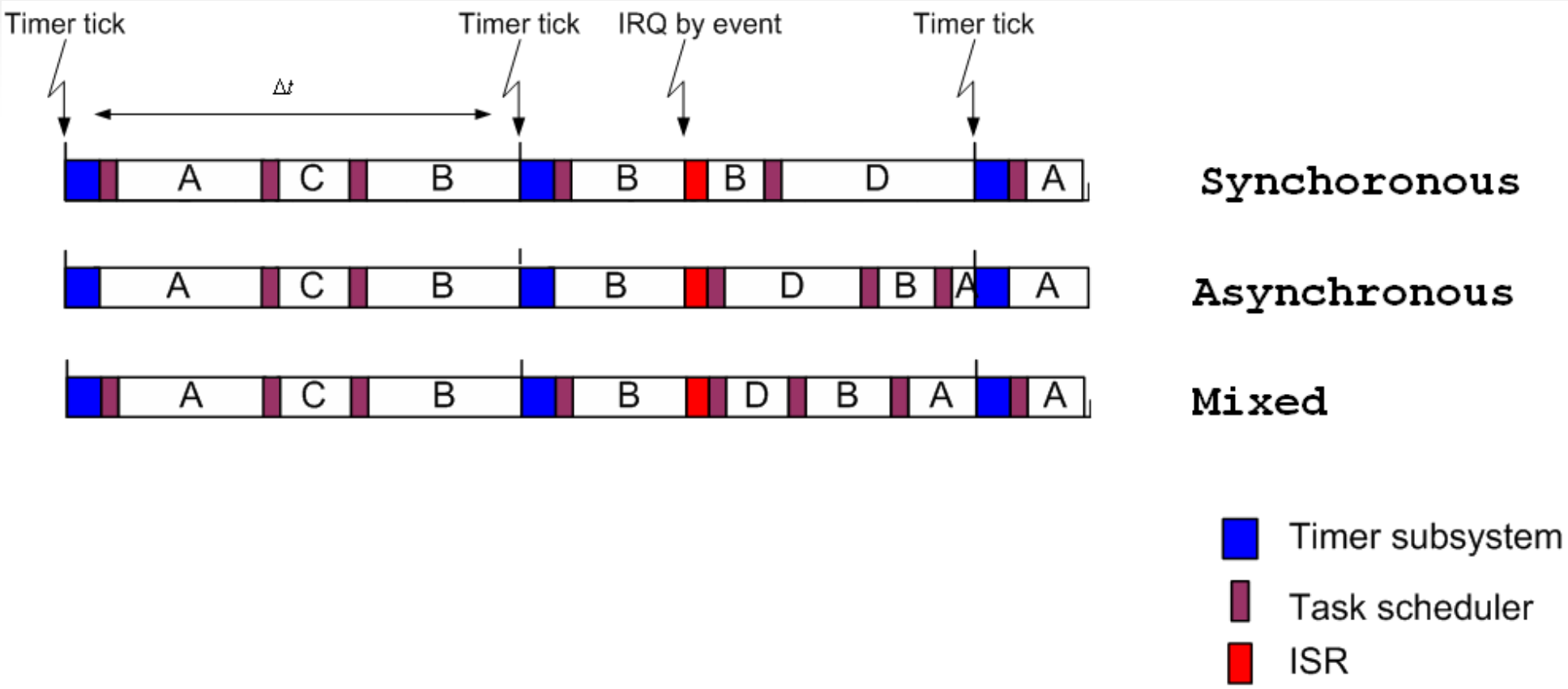
Абсолютна



Priority scheduling disciplines

- Non-preemptive (relative) discipline
  - ✓ Advantages
    - × No conflicts on resources
    - × Simpler kernel structure
  - ✓ Disadvantages
    - × Less reaction time.
- Preemptive (relative) discipline
  - ✓ Advantages
    - × Predictability
    - × Simpler kernel structure
  - ✓ Disadvantages
    - × More complicated kernel structure.
    - × Possible conflicts on resources.

Дисциплини за управление на времето



Подсистема за измерване на времето

- Тя е самостоятелен системен обект.
- Две основни функции:
  - ✓ Часовник за реално времеReal-time clock
    - × Периодична активация от импулсен генератор.
    - × Точност - Тактов генератор (между 10^5 наносекунди до милисекунди).
    - × Структура (системен часовним, броячиза сек., мин., ч., дни, месеци, години)
    - × Брояч на абсолютното време (48 ÷ 64 битов брояч, управляван от системния тактов генератор)
  - ✓ Измерване на интервали – интервални и календарни функции.

Управление на външни събития

- Същност:
  - ✓ Реализира се при апаратно прекъсване.
  - ✓ Управлява се Диспечер на събитията (Event Manager)
  - ✓ Диспечер на събитията се състои от функции за управление на събития (Event Handlers)
  - ✓ Функциите за управление на събития се активизират от функции за управление на прекъсванията(Interrupt Service Routines - ISR)
- Реализация:
  - ✓ Сигнализират се една или повече задачи за случилото се *или*
  - ✓ Разпространение (Broadcasting) на съобщение една или повече задачи

Взаимодействие на задачите

- Основни проблеми:
  - Синхронизация – изисква за задачите и нишките да се гарантира:
    - коректно изпълнение на критичен код
    - гарантиране на достъп до ресурс по определен ред
    - сигнализация при настъпило събитие
  - Комуникация – да се реализира високо и ниско ниво на междупроцесна комуникация.

Методи за синхронизация

- Основен проблем: ‘Състезание на сигнали’.
- Основни механизми:
  - ✓ Събитийни флагове (Event flags) – разпространяват събитията
  - ✓ Семафори – синхронизация и изключване
  - ✓ Мутекси (Mutex) – управление на достъпа до ресурс
  - ✓ Забрана на прекъсванията (Interrupt disabling) - управление на достъпа до ресурс
  - ✓ Променливи на състоянието (Conditional variables) - управление на достъпа до ресурс

Събитийни флагове

- Същност
  - Бълева променлива, достъпна от всички процеси /нишки, които трябва да се сигнализират/синхронизират.
  - Използвана за сигнализация ‘един-към-много’.
- Операции:
  - *set_flag , read_flag, reset_flag, read_and_reset_flag*
- Проблеми:
  - Обща памет или системни ресурси, достъпни от всички.
  - Кой начално инициализира флага при ‘един-към-много’ ?
  - Състезание на сигнали

Семафори

```
int counter;
ProcessIDqueue[ ];

up()
{
    counter++;
    if ( counter > 0 ) return;
    get first processID  from ProcessIDqueue and wakeup the process.
    return;
}

down()
{
    counter--;
    if ( counter >= 0 ) return;
    put in ProcessIDqueue ID of calling process.
    sleep_the_process();
}
```

Двоични (сигнални) семафори

- Брояч: *-1, 0, 1*.
  - Начално състояние:
    - *counter* = 0
  - Сигнализирно състояние:
    - If *counter* = 1 : включен
    - If *counter* = 0 : не е включен и никой не чака за него
    - If *counter* = -1 : не е включен и някой чака за него
  - Операции:
    - down() → signal()
    - up() → wait()

Ресурсни семафори

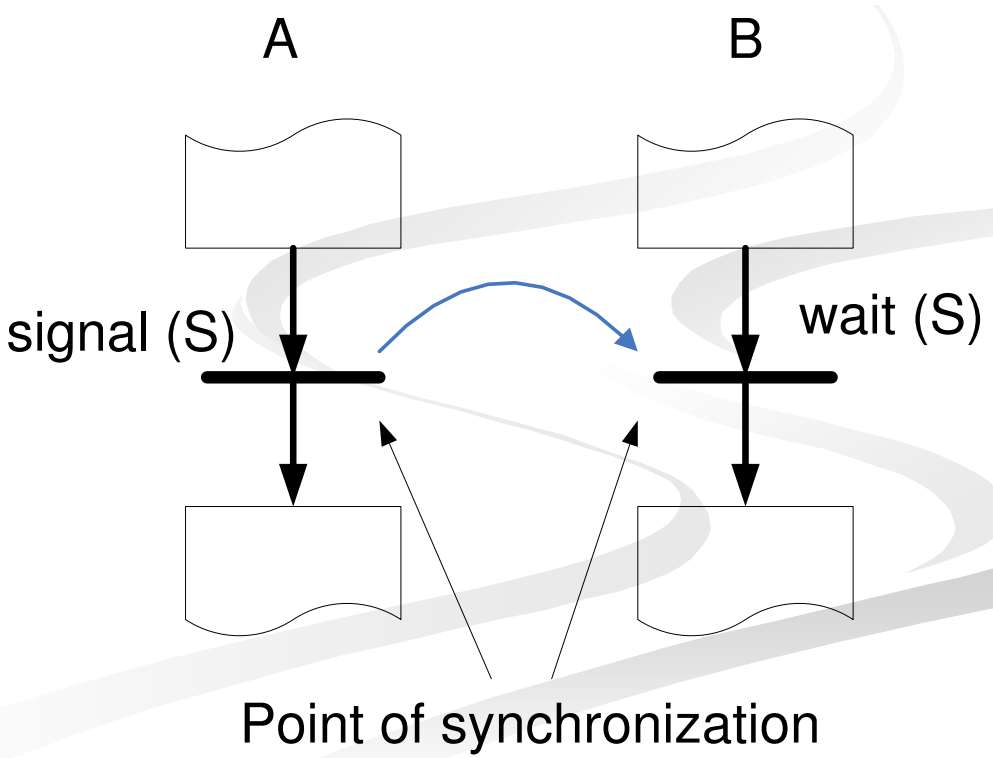
- Еднослотни:
  - ✓ *counter = { 1, 0, -1, .., -n }, **n** -> макс. брой на чакащите*
  - ✓ Начална стойност: *counter = 1*
- Многослотни
  - ✓ *counter = { K, 0, -1, .., -n }, **K** -> брой на слотовете, **n** -> макс. брой на чакащите*
  - ✓ Начална стойност: *counter = 1*
- Операции:
  - ✓ *down() → lock()*
  - ✓ *up() → unlock()*

Взаимни изключвания (Mutual exclusions)

- Същност
  - ✓ Неконкурентен достъп до ресурсите.
  - ✓ Основната задача е да изключи състезанието на сигналите.
- Видове:
  - ✓ Забрана на прекъсванията (Interrupt disabling)
  - ✓ Инструкции от вида ‘Read_Modify_Write’
  - ✓ Ресурсни семафори
  - ✓ Мутекси (Mutex)
  - ✓ Монитори (Monitors)

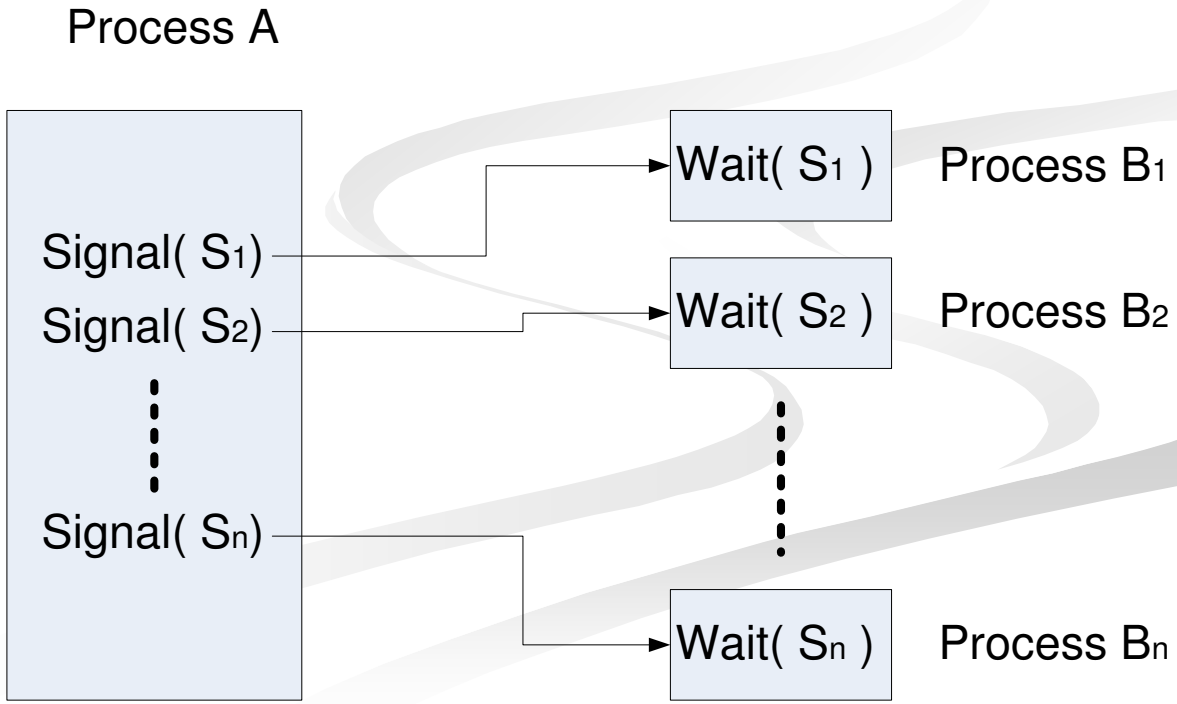
Сигнализационни схеми

- Един-към-един : използват се двоични семафори



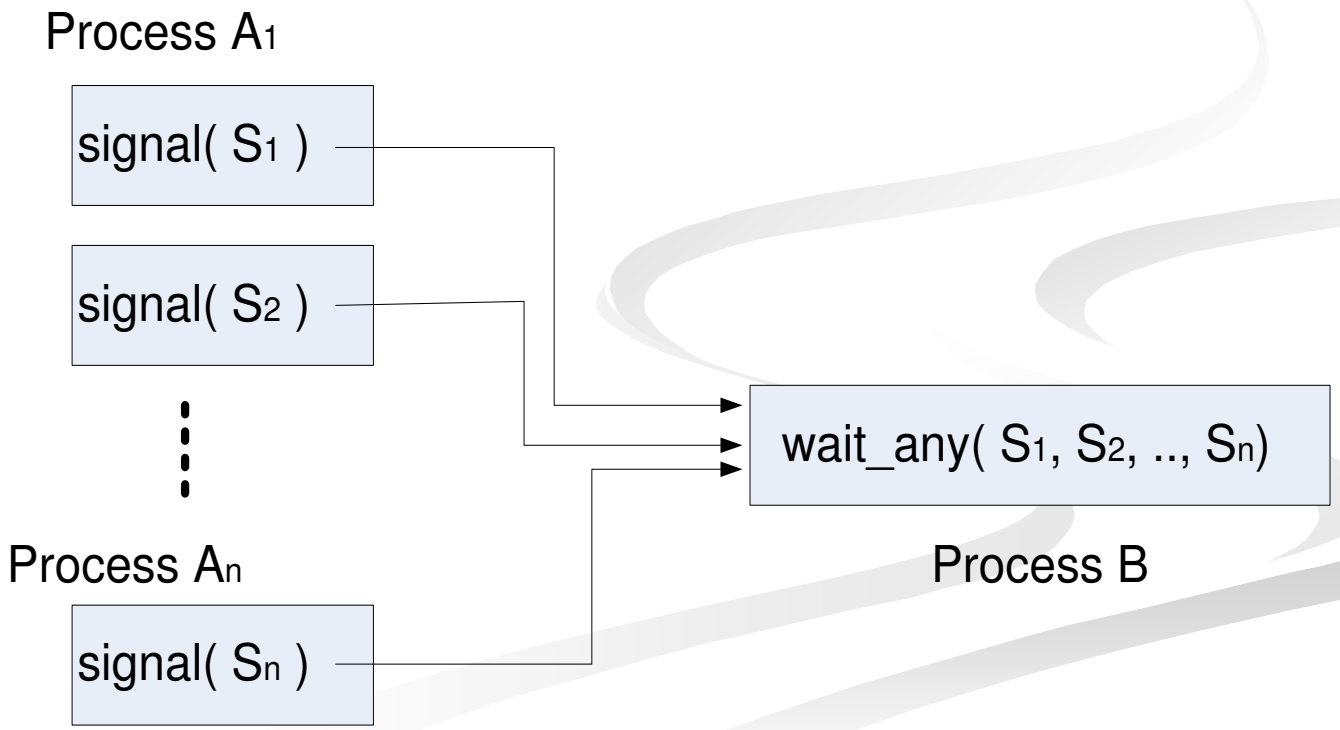
Сигнализационни схеми

- Един-към-много
  - ✓ Изчакващите процеси не се сигнализират едновременно, т.е. Не е много полезен начин за сигнализация



Сигнализационни схеми

- Много-към-един

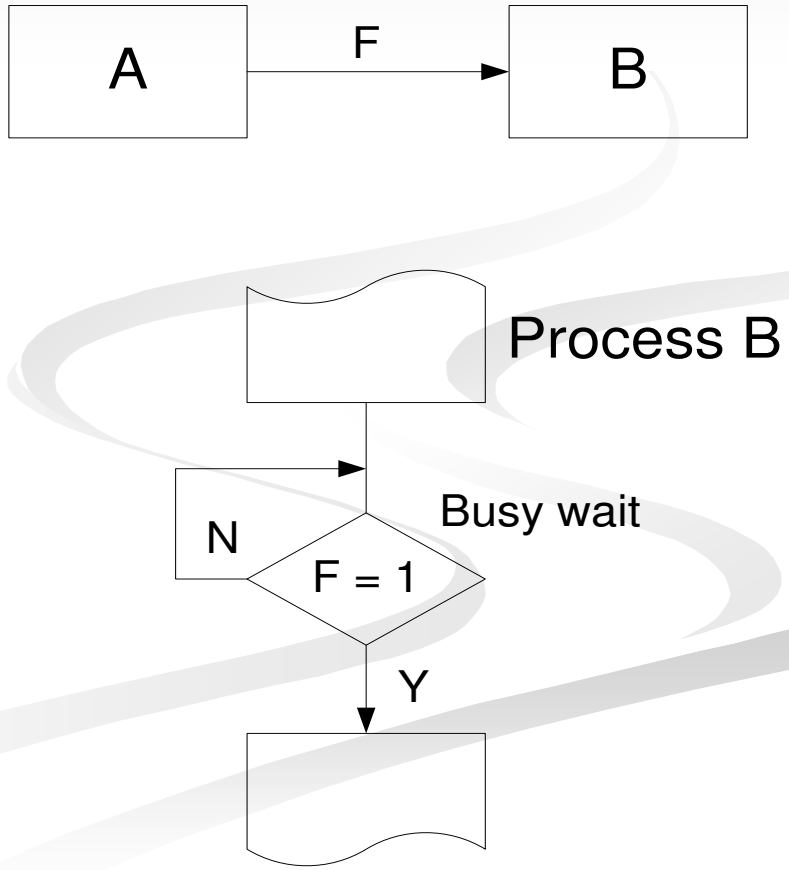


Сигнализационни схеми

- Неблокитаща сигнализация
  - Нова семафорна операция:
    - *int test_and_reset()* – връща текущата стоойност и го инициализира с **0.**
- Изпъление на синхронизационните схеми
  - Сигнализация за събитие
  - Двупосочна синхрониаация
  - Ресурсна синхронизация

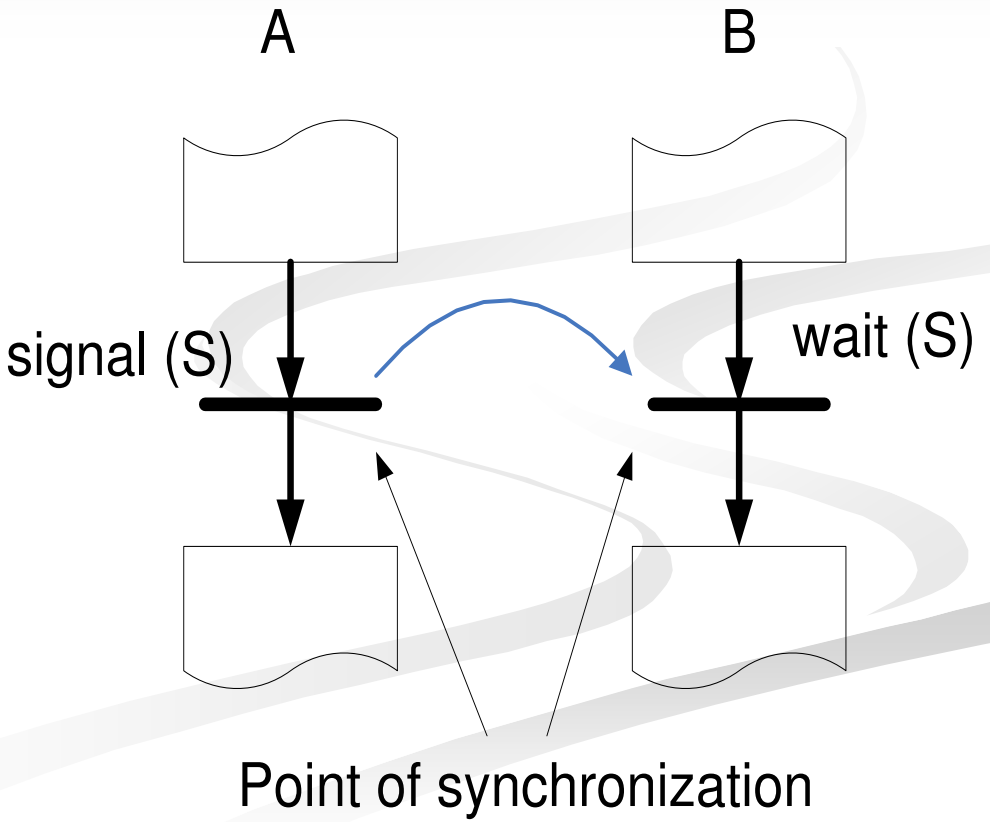
Сигнализация за събитие

- Чрез събитийни флагове



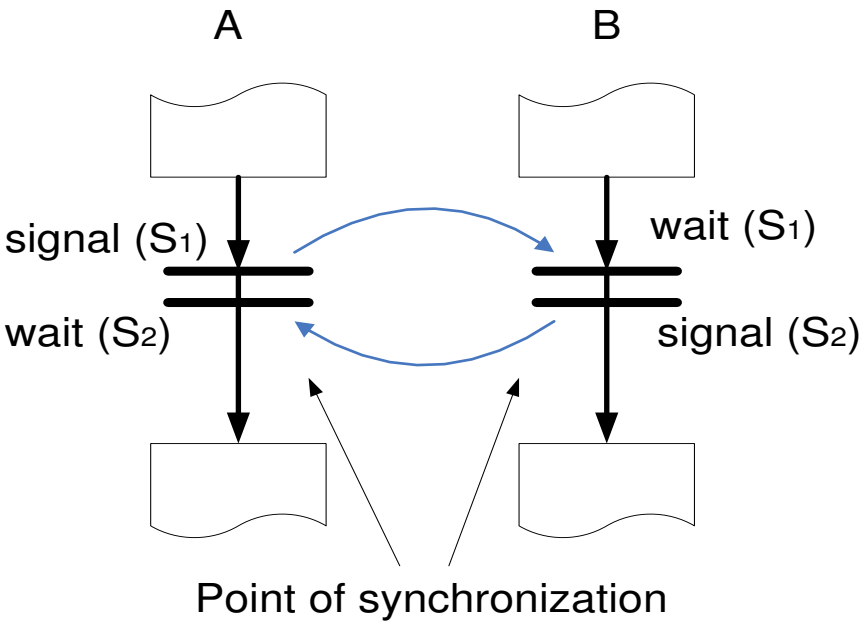
Сигнализация за събитие

- Чрез сигнални семафори

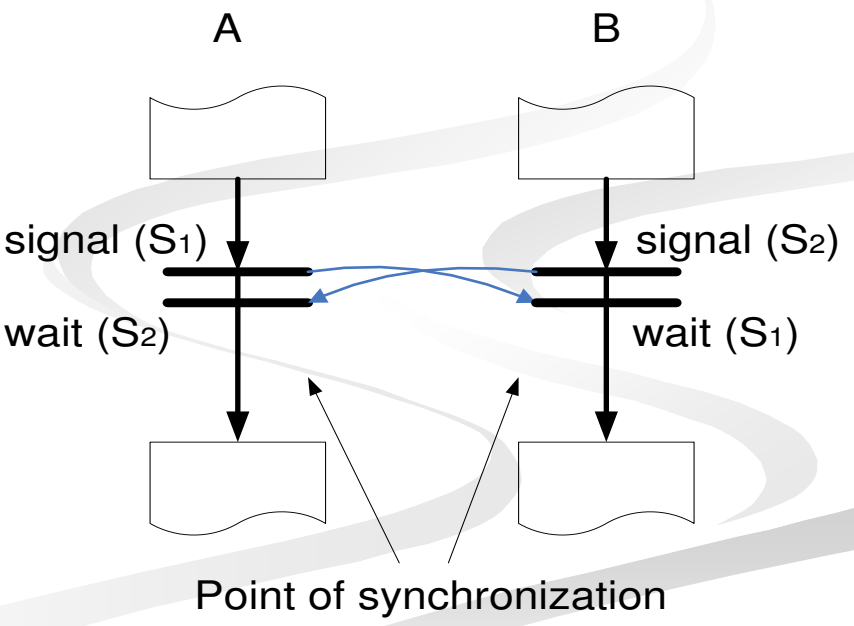


Двупосочни синхронизации

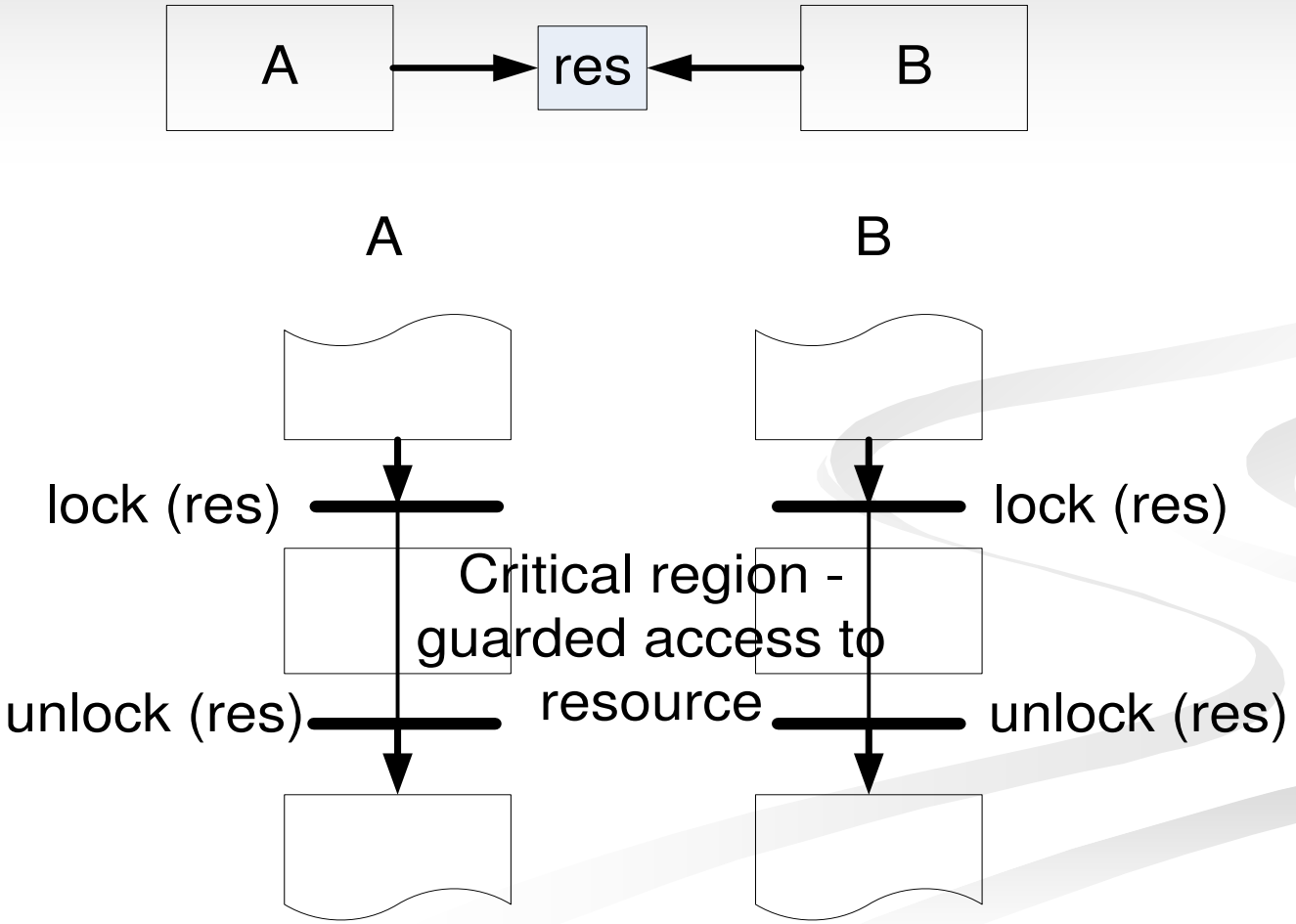
Hand-shaking



Symmetrical Rendezvous

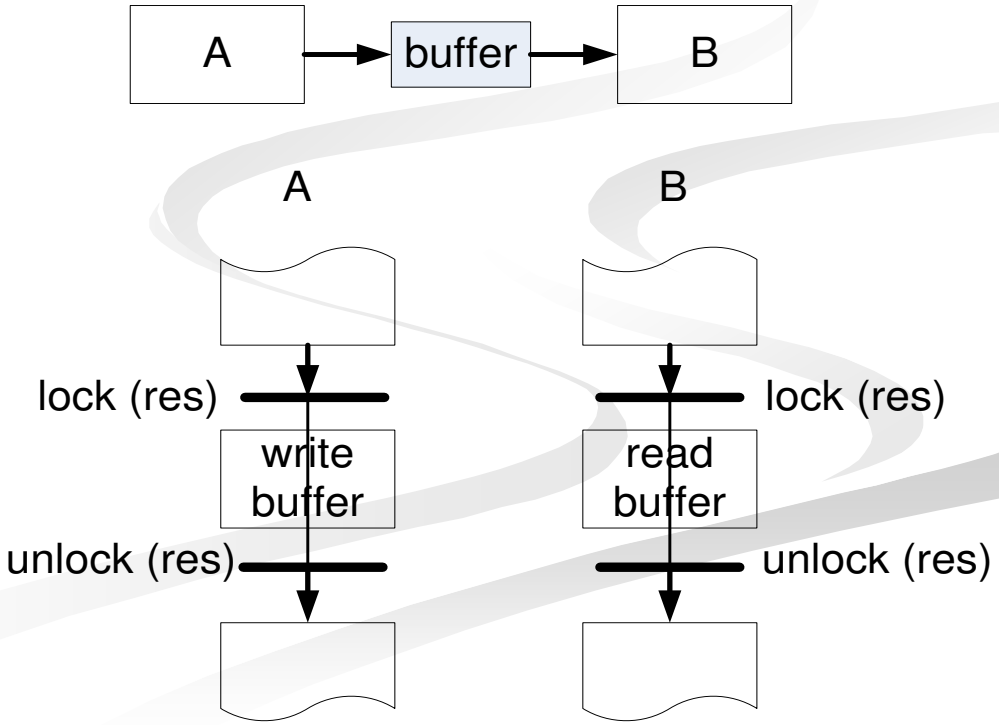


Синхронизация в ресурсни семафори



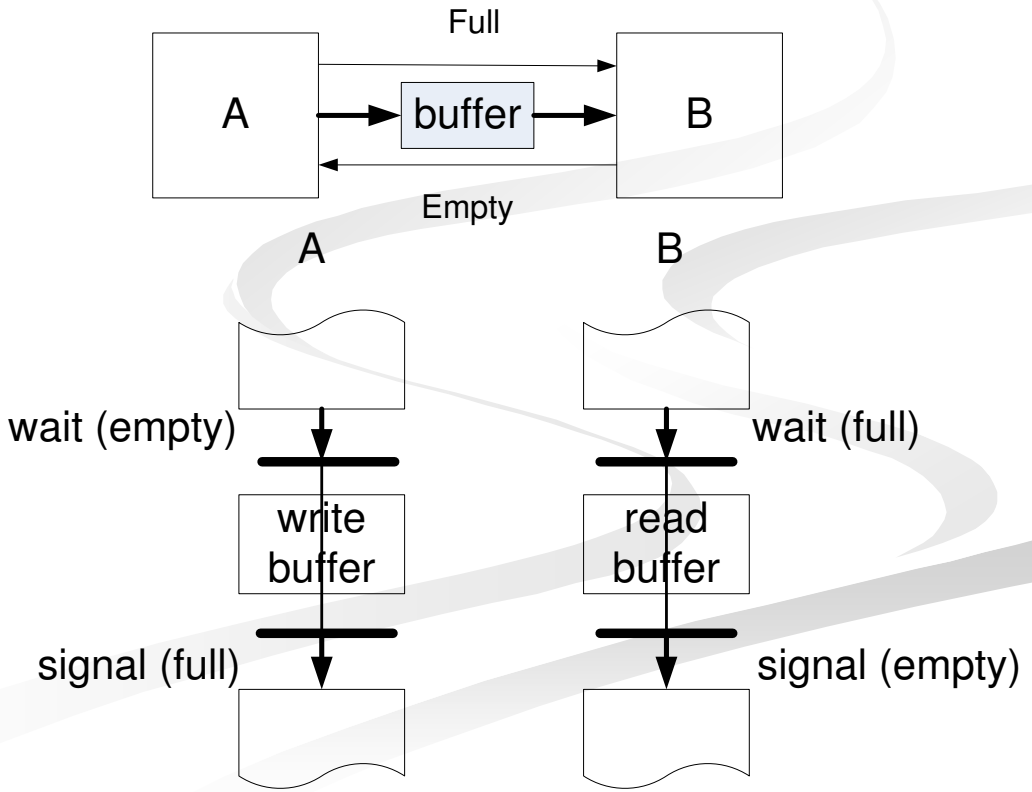
Междузадачна комуникация

- Асинхронна комуникация:
  - ✓ Основен проблем: ‘Производител-потребител’ (‘Producer-Consumer’).



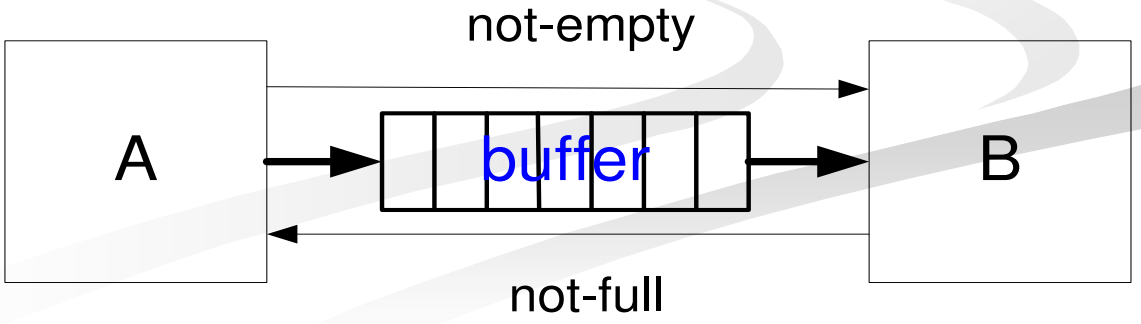
Task Interaction: Communications

- Синхронна комуникация:
  - ✓ Подобрен вариант на ‘Производител-потребител’.



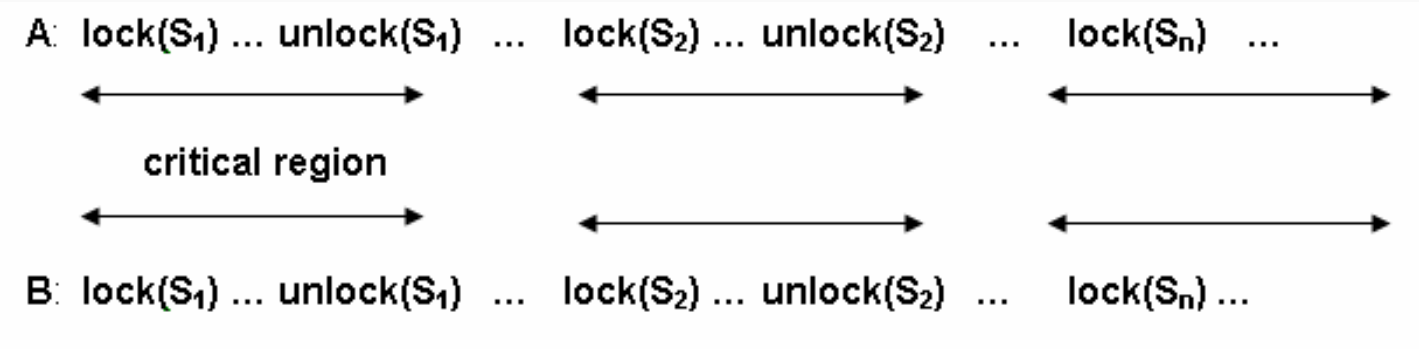
Междузадачна комуникация

- Асинхронна комуникация с многопозиционен (многослотен) буфер
  - Работата с буфера се управлява с ресурсен семафор.
  - Състоянието се сигнализира с два двоични семафора:
    - ‘not-full’
    - not-empty’

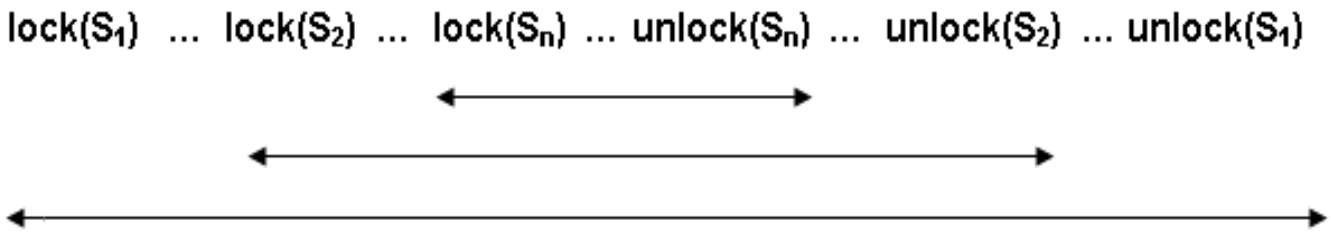


Достъп до ресурсите

- Последователен достъп



- Вложени критични секции



Взаимна блокиворка (Deadlocks)

- Тип ‘Един-към-един’

Задачите A and B се нуждаят от ресурсите R_1 и R_2.

A: *… lock (R_1) .*. lock (R_2) .. unlock (R_2) ..unlock (R_1)*
B: *… lock (R_2) … lock (R_1) .. unlock (R_1).. unlock (R_2)*

Ако P(B) > P(A) и **B** прекъсна **A** в маркирания със ‘ * ’ момент, задачите ще се блокират завинаги: **A** чака $\mathbf{R_2}$, **B** чака $\mathbf{R_1}$.

Взаимна блокиворка (Deadlocks)

- Транситивна блокировка

Задачите A, B, C използват ресурситеR_1, R_2, R_3.

A: … *lock* (R_1) .* . *lock* (R_2) .. *unlock* (R_2) .. *unlock* (R_1)

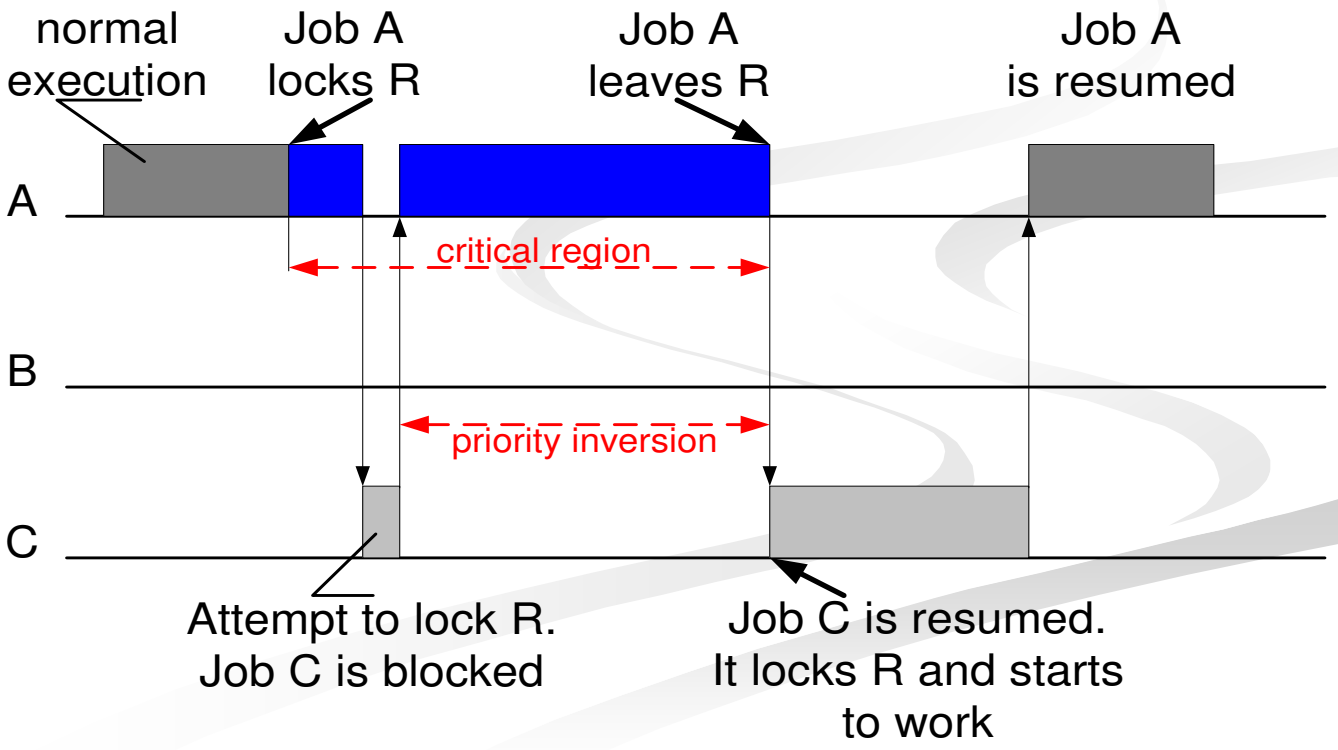
B: … *lock* (R_2) .* . *lock* (R_3) .. *unlock* (R_3) .. *unlock* (R_2)

C: … *lock* (R_3) .* . *lock* (R_1) .. *unlock* (R_1) .. *unlock* (R_3)

P(C) > P(B) > P(A), **B** прекъсва **A, C** прекъсва **B.**
Блокировката е постоянна, защото **A** чака $\mathbf{R_2}$, **B** чака $\mathbf{R_3}$, **C** чака $\mathbf{R_1}$.

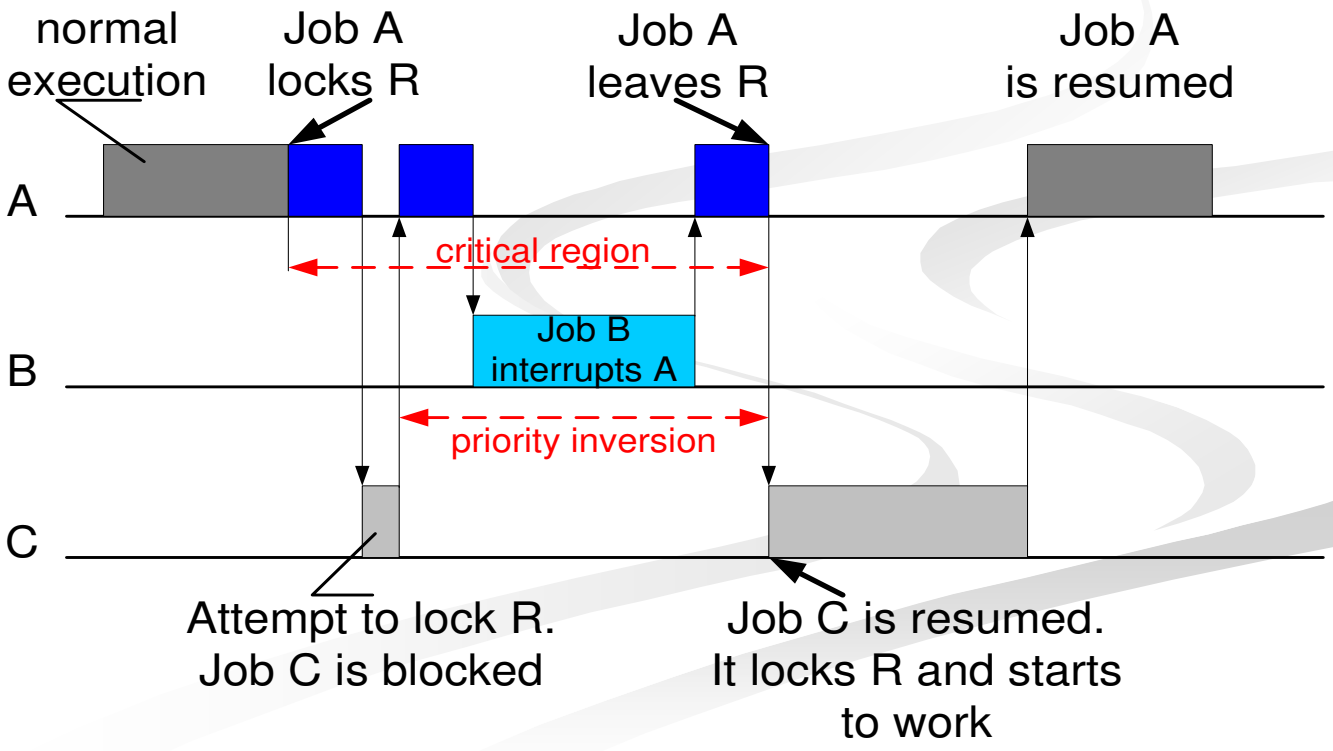
Приоритетна инверсия

- Ограничена (Bounded) инверсия
  - ✓ P(A) < P(C), ресурсът е **R**, C чака A да напусне критичната си секция и така спата приоритета.



Приоритетна инверсия

- Неограничена инверсия
  - ✓ P(A) < P(B) < P, A и C искат R, но B прекъсва A.



Анализ на диспечируемостта

- Основни решения
  - ✓ Периодични задачи
  - ✓ Спорадични задачи
  - ✓ Разширен анализ
- Диспечиране на периодичнизадачи
  - ✓ Rate-monotonic scheduling (RMS)
  - ✓ Earliest-deadline first (EDF)
- Диспечиране на спорадични задачи
  - ✓ RMS (aperiodic-servers).
  - ✓ Dual-priority scheduling

Разширен анализ

- Включва:
  - ✓ Влияние от ОС
  - ✓ Влияние от ISR
  - ✓ Процедури за стартиране на задачи
  - ✓ Процедури за завършване на задачи
  - ✓ Взаимоотношения между задачи (изчакване на критични секции)
  - ✓ Други

Въпроси ?

The Defects

The defects

- There are really three types of defects:
  - A ***failure*** is visible to the end user of a program.
    - Example: the program is supposed to print a one and instead prints a zero.
  - A ***fault*** is the underlying state of the program at runtime that leads to a failure.
    - Example: the program might display the incorrect output because the wrong value is stored in a variable.
  - An ***error*** is the actual incorrect fragment of code that the programmer wrote.
    - This is what must be changed to fix the problem.
- A failure implies at least one fault, and a fault implies at least one error, but the reverse is not true.

Syntax Versus Semantics

- Most writing on programming errors distinguishes between two basic types of errors:
  - Syntax errors involve the precise definition of how valid programs are formed.
  - Semantic errors can be broken down into:
    - Runtime errors cause the program to crash or abort in some way.
    - Logic errors cause a program to run to completion, but produce the incorrect output or result.
- Arguably, another class of errors exists:
  - Linker errors: the compiler cannot produce a binary
    - They can occur when an external variable or function name has a typo, or if there is a problem with how the compiler is installed, or for various other reasons that are outside the scope of this book.
    - These errors are language- and system-specific, and (like syntax errors) result in an error being reported to the user.

Syntax Versus Semantics

- Examples: *the divide by 0 defect*
  - ✓ In some languages, certain defect always cause a runtime error (using an incorrect array index is one example), but in others, they usually result in silent logic errors.

```
int compute_average(int array[], int count) {
    int j, total = 0;
    for (j = 0; j < count; j++) {
        total += array[j]);
    }
    return total / count;
}
```

Defect Classification Systems

- Many articles attempt to classify defects into types
  - ✓ In fact, there have been roughly as many classifications as articles.
- The Donald Knuth categorization (in book "The Errors of TeX“)
  - ✓ He grouped the changes into 15 categories: 9 for defects and 6 for enhancements; each assigned a letter of the alphabet for reference.

The Knuth categorization

- Algorithmic Anomalies
  - ✓ The code correctly follows the intent of the programmer, but the intent was wrong.
- Blunders
  - ✓ “Thinking the right thing, but writing it wrong.“: the algorithm was correct, but the code did not implement it correctly.
- Data Disasters
  - ✓ Data was incorrectly modified in some way such that the result did not reflect the programmer's intent.
- Forgetfulness
  - ✓ A simple error of omission; leaving out some code so that the program did not do all that it was supposed to do.

The Knuth categorization

- Language Lossage
  - ✓ Errors related to misunderstanding or not considering the specific features of the syntax, such as the precedence of operators.
- Mismatches
  - ✓ Calling a subroutine with incorrect parameters in a way that the compiler won't report an error.
- Robustness
  - ✓ Crashing on bad input data, reporting uninformative error messages, and the like.
- Surprises
  - ✓ Unforeseen interaction between different sections of the program.
- Typographic Trivia
  - ✓ Simple errors when typing in the program.

The Adam Barr categorization

- This classification has 4 main groups because some of Knuth's categories are merged and other are ignored.
  - ✓ It ignore the enhancement categories.
  - ✓ It doesn't use Knuth's categories 'Mismatches', 'Robustness', and 'Language Lossage'.
    - ✗ In the real world, mismatches do occur, but usually only in large programs, especially when calling functions written by someone else.
    - ✗ Knuth gave Robustness its own category because TeX is written specifically to process input and produce output, as a compiler would, so handling incorrect input and producing good error messages for the user is important.
    - ✗ Many times there are no defects that would fall under Language Lossage, although in the real world, these can be common.

The Adam Barr categorization

- The categories are:
  - A-Algorithm
  - D-Data
  - F–Forgotten
  - B–Blunder
- Categories are broken into subcategories
  - The subcategories are identified with the notation *C.subcategory*, where C is the initial of one of the main categories (A, D, F, or B) and subcategory is a descriptive name.

A–Algorithm group

- A–Algorithm = ‘Algorithmic Anomalies’ + ‘Surprises’
  - The distinction between the two is somewhat difficult anyway and seems to be primarily related to the distance in the code between the error and the fault, or the fault and the failure.
- Subcategories:
  - A.off-by-one
  - A.logic
  - A.validation
  - A.performance

A.off-by-one

- It occurs when the code performs a calculation or includes an expression that is one away from what it should have been, which results in:
  - ✓ the code processing the wrong number of pieces of data, or
  - ✓ returning a value that is incorrect, or
  - ✓ taking a branch in the code at the wrong time.

A.off-by-one

- Examples
  - ✓ The fencepost error:
    - ✘ It occurs when the number of elements is miscounted because of neglecting to account for the final element
  - ✓ The wrong comparison operator, confounding < and <= or > and >=.

```
// The following code, which tries to check if someone is old enough
// to vote in the United States (you must be 18 or older to vote)

if (age > 18) {

    // OK to vote!

}
```

A.logic

- An A.logic error occurs if the programmer has designed a logically incorrect way to achieve the desired result.
- In many cases, the error involves the incorrect calculation of a value based on some data.
  - Often, this is caused by a bad assumption about the data in question.
- Examples:
  - A loop termination condition that never changes.
  - Break out of a loop at an incorrect time, or neglect to break out at the proper time.

A.validation

- Many blocks of code that need to be debugged are functions that take parameters passed from other code.
  - ✓ Often, the function documentation restricts the range of values allowed for certain parameters.
- The compiler usually does not check such restrictions (depending on how they are specified), and the question becomes whether checking the validity of arguments should be part of the function's algorithm
  - ✓ Whether parameters should be checked for validity within the function or if it is the caller's responsibility to ensure that parameters are valid before calling the function. If the function is supposed to check this, not having this code is a defect in the algorithm.

A.validation

- Example:

```
my_function((void *)0x12345)
```

```
void my_function(char * my_string) {
    if (my_string == NULL) {
        return;
    }
    if (strlen(my_string) == 0) {
        return;
    }
}
```

A.performance

- Performance problems occur when a program performs its task properly, but uses far more resources than it needs.
  - ✓ Often, the resource is time, but it could also be memory, disk space, network packets, or any other limited resource.
- Because different programmers can solve the same problem with different algorithms and one of them is likely faster than the other, the question of when inferior performance becomes a bug is highly subjective.
  - ✓ If the code is 10% larger than it could be, that is often not an issue, unless memory use is critically important.
  - ✓ On the other hand, a program that is 100 times slower than necessary is generally considered to have a performance problem, no matter what the situation.

D-Data group

- It = ‘Data Disasters’
  - ✓ This category refers more specifically to cases where the code reads or writes incorrect data, or accesses the wrong storage location.
- Subcategories:
  - ✓ D.index
  - ✓ D.limit
  - ✓ D.number
  - ✓ D.memory

D.index

- It occurs when an invalid index is used when walking through an array or other data structure
- Many languages use zero-based arrays; that is, the valid indices for an array of size ***n*** go from ***0*** to ***n-1***.
  - ✓ This leads to a common indexing error when looping through such an array, starting at ***1*** instead of ***0***.
  - ✓ Similarly, on the other end, going past the end of the array – ending at ***n*** instead ***n-1***

```
for (i = 0; i <= n; i++)
    // code that processes array[i]
```

```
for (i = 1; i < n-1; i++)
    // code that processes array[i]
```

D.limit

- A D.limit error involves failing to process data correctly at the limits: the first or last element of the data set (or possibly, the first few or last few elements).
  - An index error often leads to a limit error.
  - It may cause the code to crash accessing past the end of the data - if the indexing is too expansive.
- A D.limit error occurs when the code makes assumptions that are true except on the first or last element.
- A D.limit error cases where the code works incorrectly on certain inputs near the beginning or end of the range of valid inputs.
  - That is which tend to slightly misprocess all inputs, these are cases where the code works fine on most inputs, but completely fails on a small subset near the limit.

D.number

- The D.number class of data errors relates to how numbers are stored on a computer.
  - ✓ D.number errors are usually not specific to a particular language, but rather to how a particular processor stores numbers (and because different machines use the same processor, the same type of error can occur in many languages on many machines).
- The most basic of these is an overflow, which is when a program attempts to store a number in an area of memory that is not large enough.
  - ✓ One form of an overflow error is an assignment between variables of different sizes
  - ✓ Another form of overflow can happen with types of the same size: data produce larger than the maximum number that can be stored or possibly a signed/unsigned error.

D.memory

- D.memory errors can cause some of the hardest-to-find bugs in the real world
- The D.memory error involves mismanaging memory.
  - One way to cause this error is to attempt to access memory that is not accessible to the program, by improperly manipulating an array index or pointer .
  - Another way to cause this error is to allocate memory after it is freed.
  - Instead of freeing memory too soon, programs can forget to free it, which causes a memory leak.
    - Code with memory leaks often works for a long time and then fails unexpectedly when new memory cannot be allocated.
    - This is an unpredictable situation based on the hardware being used, what other applications are running, and other hard-to-predict factors.

D.memory

- D.memory errors can cause some of the hardest-to-find bugs in the real world
- The D.memory error involves mismanaging memory.
  - A final way to mismanage memory is to use the same variable for two different reasons in different sections of code, but later discover that the logical scope of the two areas overlaps.
    - This error can occur when code is cut-and-pasted into the middle of a larger section of code that uses the same variable. It can also happen when programmers reuse the same variable name for different functions on the theory that it saves memory.

F–Forgotten group

- F–Forgotten = Forgetfulness
  - This category generalized to include all control flow errors that occur because the statements in the program are not executed in the order that the programmer intended.
    - Many errors are caused by the programmer wanting the computer to "Do what I mean, not what I say."
- Subcategories:
  - F.init
  - F.missing
  - F.location

F.init

- One of the most common types of instructions to leave out are ones that initialize variables.
  - Many variables are not initialized when they are defined, and many languages do not assign a default value in this case, which results in the variable containing whatever value happens to be in the memory location the variable is stored at.
  - An uninitialized variable becomes a defect when it is actually used in code that expects it to be initialized.
    - Usually, this error happens because certain paths through the code avoid the initialization, although in some cases, the variable is never initialized.
    - The more unusual the uninitialized path is, the less likely such a defect will be found early.

F.missing

- Other statements besides initialization can be left out
  - The case of missing initialization is just so common that it deserves its own subcategory. F.missing is the general case.
- Programmers can neglect to type in a statement, or accidentally delete it while moving code around, cleaning up comments, and so on.
  - Finding these types of bugs can be tricky unless a comment makes it obvious

F.location

- The F.location refers to code that is in the wrong sequence within the program.
  - One example is initializing a variable inside a loop rather than outside it.
  - The sequence of instructions does not match the intended order of operations.
    - Often, two instructions are swapped.
  - Instructions can be placed in the wrong block of code, particularly if blocks are nested several layers deep.

B–Blunder group

- B–Blunder = ‘ Blunders’ + ‘Typographic Trivia’
  - ✓ This category exists because Knuth wrote his code on paper and then typed it, so he made occasional transcription errors in the process.
  - ✓ Because many programmers type code directly into the computer, they would never encounter that problem.
- Subcategories:
  - ✓ B.variable
  - ✓ B.expression
  - ✓ B.language

B.variable

- An easy and common mistake is using the wrong variable name:
  - One source of B.variable errors is cutting-and-pasting similar code.
  - Anywhere a variable is used, it's possible to use the wrong one - on the left or right side of an assignment, as an argument to a function, as a return value from a function, and so on.
    - One situation that arises is switching two variables in the parameters to a function, which passes unnoticed by the compiler as long as they are the same type.

B.expression

- The B.expression is a more general form of the B.variable.
  - A variable on its own is an "expression," but the case of just using the wrong variable name is so common that it was separated.
  - B.expression covers other cases in which expressions are incorrect not because of the algorithm being wrong, but because of a momentary cramp in the programmer's brain.

B.expression

- Examples:
  - The most basic of these errors is when the code uses the wrong operator
  - Given that expressions can be arbitrarily complex, there is an arbitrary opportunity to make mistakes.
    - One place that bad expressions may appear is in an if statement
  - The logical "and" and "or" operators are common areas where the wrong operator is used in an expression, usually by using and instead of or or vice versa.

B.language

- Some languages have syntax features that can lead to improper expressions.
  - ✓ This is what Knuth calls Language Lossage.
  - ✓ For example, expressions that depend on the precedence of operators can be interpreted in a non-intuitive way.

Tips on Walking Through Code

- It is often not necessary to follow all the steps.
  - ✓ At any point, the reason for the bug may suddenly pop into your mind, even if you are not directly considering the code that contains the defect.
  - ✓ But, if that doesn't happen along the way, hopefully by the time you finish the final step, the bug will have revealed itself.
- The steps are as follows:
  1. Split the code into sections with goals.
  2. Identify the meaning of each variable.
  3. Look for known "gotchas."
  4. Choose inputs for walkthroughs.
  5. Walk through each section.

Split the Code into Sections with Goals

- The first step to understanding the code is to split it into sections and identify the goals of each section.
- A section is a snippet of code that accomplishes a specific task.
- The "goal" of a section of code is the set of changes that the code is intended to make to the data structures used by the program.

Identify the Sections in the Code

- Most functions:
  - ✓ begin with introductory code to handle special cases, deal with errors, and so on,
  - ✓ and end with code that cleans up and possibly returns values to the calling function.
  - ✓ In between these is the code that implements the main algorithm.
- It is useful to note where the introductory code ends and where the cleanup code begins. Mark the area between those as the location of the main algorithm.

Identify the Sections in the Code

- The most basic step is to locate the main part of the algorithm.
  - ✓ The main algorithm is the part that you would talk about if you were telling someone what the code did.
  - ✓ Tricky input-specific bugs might hide in the main algorithm: this is the part that actually corresponds to the mathematical algorithm that the code implements.
- The introductory and cleanup code can still harbor bugs and need to be checked as carefully as any other piece of code.
  - ✓ However, it is true that the introductory and cleanup code usually execute on any input, so they're tested all the time.

Identify the Sections in the Code

- If the main algorithm consists of more than just a few lines of code, it needs to be split into smaller sections.
  - Again, consider how you would describe the algorithm to someone else
    - Each part of that description is probably one section.
  - If you would describe an algorithm as "first read in the data, then organize it by key, then output it," you would try to separate the code into those three sections.

Identify the Sections in the Code

- Example:

```
int find_largest_hash(String s[]) {
    if (s.length == 0) {
        throw new InvalidParameterException();
    }

    HashCalculator hb = new HashCalculator();
    int largesthash = hb.hash(s[0]);
    int newhash;

    for (int j = 1; j < s.length; j++) {
        newhash = hb.hash(s[j]);
        if (newhash > largesthash) {
            largesthash = newhash;
        }
    }

    hb.flush();

    return largesthash;
}
```

Identify Goals for Each Section

- After you split the code into sections, identify the goals of each section.
  - ✓ At the end of the section, what variables should be modified and how?
  - ✓ What invariant conditions should be true?
  - ✓ How should the data structures be set up?

Identify Goals for Each Section

- When you have finished mentally dividing the code into sections with goals, check that each goal is well contained.
  - ✓ Code that starts working on the next goal before it logically finishes a previous one can be prone to bugs.
- For ***if*** statements, try to state the goal of the if condition itself
  - ✓ "The if() block will execute if the user has not been validated yet."

Identify Goals for Each Section

- If a section of code is a loop, you need to determine the overall goal of the loop.
  - ✓ However, you should also try to determine the goal of the loop after one iteration.
- Some languages allow ***assert*** statements, which are logical expressions that cause the program to halt if they are false.
  - ✓ The gaps between sections are often a good place to put assert statements that verify if the goal of a section was properly achieved

Comments

- Comments are an important part of determining the goal of a piece of code.
  - ✓ They represent the only chance a programmer has to communicate his or her ideas in plain language.
- Many programmers write comments as hints for when they come back to look at the code.
  - ✓ In many cases, long comments indicate areas that the original programmer felt were tricky, unclear, or in some other way unlikely to be obvious upon later viewing.
  - ✓ The presence of such comments usually indicates the location of the key parts of the algorithm.

Comments

- Often, comments can also help identify useful sections within the code, because many times, a multiline explanatory comment precedes a block of code worth grouping into one section, and the comment tries to explain the goal of the code.
- Some comments are done by rote, in the apparent belief that mundane operations need a comment.
  - ✓ These types of comments are unlikely to highlight buggy areas. On the other hand, a simple comment like the following, which is obviously wrong, is a sign that significant changes might have been made to the code since it was originally written
- However, it is important not to let comments mislead you.

Identify the Meaning of Each Variable

- After you identify the goal of each section, look at the variables used in the code and identify the "meaning" of each one.
  - The meaning of a variable refers to what value, conceptually, it is supposed to contain.
- Five steps:
  - Variable Names
  - Look at the Usage of Each Variable
  - Restricted Variables
  - Invariant Conditions
  - Track Changes to Restricted Variables

Variable Names

- Variable names, like comments, can be both useful and misleading.
- Unlike sections of code, all variables have names, which can usually be counted on to provide some hint of the variable's meaning.
  - A variable's name is like a miniature comment from the programmer that appears every place the variable is used.
  - As with comments, however, you have to make sure that the variable really is used the way the name indicates.
  - Furthermore, some variables, even important ones, have single-letter or other uninformative names

Variable Names

- Variable names, like comments, can be both useful and misleading.
- Unlike sections of code, all variables have names, which can usually be counted on to provide some hint of the variable's meaning.

```
float average_balance;   // good

string name;             // OK, but name of what?

int k;                   // unclear; could be anything
```

Variable Names

- Unlike comments, a compiler or interpreter does not completely ignore variable names, because a variable name refers to a specific piece of storage.
  - ✓ But the compiler or interpreter doesn't care about the actual name - naming a variable ***a***, ***total***, or ***wxyz*** won't affect how the compiler treats it.
  - ✓ What matters is that a variable is properly declared, defined, and used throughout the program.
- If a variable has an unclear name or a name that does not match its real meaning, you should try to come up with a new name, or at least a verbal definition of the meaning.

Look at the Usage of Each Variable

- For each variable used in the function or block of code, see where it is used.
  - The first step is to distinguish where the variable is used in an expression - and therefore does not change - from where it is modified to hold a new value.
    - This is not always obvious; some variables, especially data structures, can be modified inside of the functions they are passed to as arguments.
  - Some languages have ways to indicate that a variable will not be changed inside a function (such as the const qualifier in C and C++), but these are not always used.

Look at the Usage of Each Variable

- After you determine where a variable is modified, you can start to understand how the variable is used.
  - ✓ Is it constant for the entire length of the function? Is it constant in one section of code?
  - ✓ Is it used only in one part of the code, or everywhere?
  - ✓ If it is used in more than one part, is it merely being reused to save declaring an extra variable (loop counters are often used this way), or does its value at the end of one section remain important at the start of the next section?

Look at the Usage of Each Variable

- When looking at loops, think about the state of each variable at the end of the loop. Separate the variables into:
  - ✓ those that were invariant during the loop
  - ✓ those that were used only during the loop (such as variables used to hold temporary values), and
  - ✓ those that will be used after the loop code with an expectation about their value (based on what happened during the loop).
- A loop counter can fall into either of those last two categories:
  - ✓ it is only used to control the loop;
  - ✓ it is used after the loop is done to help determine what happened in the loop (in particular, if it terminated early).

Look at the Usage of Each Variable

- Because the return value of a function is important, note whether a variable is used temporarily inside a function, or if it is actually going to be part of the data returned to the caller of the function.
- Make sure that all variables are initialized before they are used (some compilers and interpreters warn you if this is not the case).
  - ✓ Many variables are not given an initial value when they are defined, so it is important that those variables are assigned a value, in all possible code paths, before they are used in an expression.

Restricted Variables

- Restricted variables can only hold a particular subset of the values that they would normally be allowed to hold based on their type.
  - Consider this part of the variable's meaning.
- Some languages allow such variables to have their restrictions explicitly stated, but often, programmers don't take advantage of this even where it is available.
  - There is often a tradeoff between strict type-checking and ease of programming.

Restricted Variables

- Ideally, any restricted variable would be identified as such - at least in a comment when it is defined, possibly in the name of the variable itself.
  - ✓ Restricted variables are often used in ways that cause errors if the variable ever contains a value outside of its intended set.
  - ✓ It is important to determine if and how a variable is restricted.

Restricted Variables

- An array index is a form of restricted variable because the proper values are defined by the size of the array.
- Some languages check array access at runtime and generate an error; other languages silently access whatever memory the index winds up indicating.
  - The runtime error is preferable because it makes it apparent that something is wrong, but both errors can occur for the same reasons.
- Unfortunately, the size of an array can itself be dynamic and difficult to determine at a given point in the code. Furthermore, an array can be indexed using a complicated expression.

Invariant Conditions

- Invariant conditions are a more general form of a restricted variable.
  - It is an expression, involving one or more variables, that is supposed to be true at any point during the execution of the program, except for brief moments when related variables are being updated.
  - It is usually a convention established by the programmer based on how he or she wants to manage the data structures used by the program.

Invariant Conditions

- When considering a variable that is a nontrivial data structure, try to think of any invariant conditions that will be true if the data structure is in a consistent state.
  - ✓ Make sure that all the relevant parts of the data structure are initialized if needed.
  - ✓ When the data structure is changed, ensure that the invariant conditions are still satisfied.

```
if ((list_head != NULL) && (list_head->next != NULL))

    (list_head->next->previous == list_head)
```

Invariant Conditions

- You need to note invariant conditions where they exist because they constitute an implicit goal before and after every block of code in the program.
  - Because goals are a theoretical idea that the compiler or interpreter is not actively concerned about, invariant conditions are also good candidates for including in ***assert*** statements for languages that support them.
- The previous statement about invariant conditions being true "except for brief moments when related variables are being updated" is important.
  - For multithreaded programs, take care that those "brief moments" are synchronized, so that another thread won't find the variables in a state where the invariant condition is false.

Track Changes to Restricted Variables

- Some variables' restrictions are usually logical rather than enforced by the compiler or interpreter.
  - ✓ It is important to check modifications to the variable to make sure that the value remains properly restricted.
- Modification of restricted variables can be checked with an inductive process.
  - ✓ That is, before a variable is modified, if you assume that the current value is properly restricted, it is possible to prove that the value after modification is properly restricted.
  - ✓ If you can show that the variable is initialized with a proper value, and that every modification keeps the variable properly restricted as long as it is properly restricted beforehand, you prove that the variable is always properly restricted.

Look for Known Gotchas

- If you have split the code into sections with goals and identified the real meaning of each variable, and nothing has jumped out as being incorrect, you can proceed to choosing inputs and walking through the code.
- First, however, you can quickly scan the code for a few "gotchas" without getting into the nitty-gritty details.
  - Loop Counters
  - Same Expression on Left- and Right-Hand Side of Assignment
  - Check Paired Operations
  - Function Calls
  - Return Values
  - Code That Is Similar to an Existing Error

Loop Counters

- Loop counters are often used to index into arrays. In languages that have zero-based arrays, notice if the check to exit a loop uses **<=** in the comparison, as opposed to **<**.
  - ✓ The comparison with **<=** might be correct but it is suspicious.
- Be aware of code that modifies the loop counter within the loop.
  - ✓ This is usually done for a reason and (hopefully) is accompanied by a comment, but it makes it more difficult for you to think through what really happens during execution of the loop - especially if the modification is only done in certain cases (depending on the contents of the data being looped through).
  - ✓ If a **continue** statement is added somewhere within the loop's loop, it would skip the code that modifies the loop counters.

Left- and Right-Hand Side of Assignment

- The same variable or expression sometimes appears on the left- and right-hand side of assignment statements that are near each other.
  - This can happen when the variable's value is used to calculate the value of another variable, and then the first variable is marked as empty, deleted, invalid, and so on.
  - Typically, there is a step where the variable is used, and a step where the variable is modified.

Check Paired Operations

- Many operations that do something in a program have a corresponding "undo" operation, which must be properly paired.
  - One common example is memory allocation, especially temporary memory allocated by a function.
    - All the temporary memory that a function allocates needs to be freed before the function exits, no matter under what condition it exits.
- Some languages do not have explicit memory allocation and deallocation, but certain other operations still must be paired up: *acquiring and releasing locks*, *adding to and subtracting from reference counts*, …

Function Calls

- Function calls can be difficult to walk through because the code inside the function is not right in front of you.
  - In the best case, you have the code for the function available, but usually, you have to trust the documentation.
- A properly written function modifies only the variables that it is supposed to modify.
  - A call to the function can be treated like a single assignment statement, although it's one that can modify multiple variables and do more complicated modifications of arrays and structures.

Function Calls

- When you look at code that calls a function, the main thing to check is that the parameters are passed correctly.
  - Most compilers and interpreters catch an argument of the wrong type being passed, but not the wrong argument of the correct type.
- One way to pass the wrong argument is when it is an index into an array. Because every element of the array has the same type, you can pass the right type, wrong argument just by botching the index.
  - Because the index is likely to be of a common type this is not difficult to do.

Return Values

- Although many functions manipulate structures that are passed in to them, for many others, the return value is what it's all about - the only permanent result of the function's execution.
  - ✓ Therefore, all the careful code that has been written and walked through will be for nothing if the function returns an incorrect value.
- The most basic mistake is simply returning the wrong variable.

Return Values

- Some functions have multiple return statements.
  - Returning from a function at the point where the result has been found is often easier than having to check if there is still more work to do.
  - It might be cleaner to have the ***return*** statement at each point where the ***return_value*** was set, instead of using the ***flag_variable*** to avoid the remaining code.
  - If you have multiple return statements, make sure that every path through the code hits one.
- Make sure the data being returned is still valid.
  - Do not return a pointer to storage that has already been freed !

Code That Is Similar to an Existing Error

- If you find a particular error that looks like it could be repeated somewhere else in the code, search for other locations where the error might have been made.
- If you discover that the code calls a function with the arguments in the incorrect order, you must check other places where the function is called.
- If a boundary error is discovered in the access to an array, check other places where the array is accessed.

Code That Is Similar to an Existing Error

- Defects do repeat themselves. This can be:
  - ✓ because code is duplicated, or
  - ✓ because the original programmer tended to make the same mistake, or
  - ✓ because of a misunderstanding about how the code worked - where the programmer was trying to consistently do the right thing, but wound up consistently doing the wrong thing.

Choose Inputs for Walkthroughs

- If you tried the preceding steps and still don't know what the bug is, you probably need to walk through the code by hand.
  - In a perfect world, you would prove to yourself that every section accomplishes its goal, that every variable sticks to its meaning, and that the proper value is returned or displayed, leaving no doubt that the function is correct for all inputs.
- In a sense, walking through the code is less than ideal.
  - It introduces an element of uncertainty because no matter how many inputs you try, the bug might not be exposed by any of them.

Choose Inputs for Walkthroughs

- Still, in many cases, the only way to unearth a defect is to walk through the code.
  - To do this, you need to select inputs to the code.
  - Except for short standalone programs that are hard-coded to calculate a given value (or set of values), all sections of code - be they a program, a function, or just a piece within a larger section of code - behave differently based on what input they receive.

Choose Inputs for Walkthroughs

- In cases where you try to track down a bug that has been reported by someone else, that person might have provided specific inputs that cause the problem to occur.
  - This is then your first candidate for a walkthrough.
  - You need to choose your own series of inputs to figure out a hard-to-reproduce or insufficiently documented bug, to check new code before releasing it, or in cases where the reported inputs are too complicated to use.
  - Walking through code is time consuming; you cannot walk through code with all possible inputs.
    - Hopefully, you can walk through with a small sample that is nonetheless representative enough of all possible inputs to expose all possible bugs.

Choose Inputs for Walkthroughs

- When you design inputs for code, remember that you are not limited to choosing only inputs to the outer function or the entire standalone program.
  - ✓ It is often easier to break the code into smaller groups and walk through them first.
  - ✓ After you are confident that these smaller groups handle various inputs correctly, you can move back and walk through larger sections of code without having to revisit the details of the sections you already checked.

Choose Inputs for Walkthroughs

- The easiest way to break up code is when your functions are layered, one on top of another.
  - ✓ Start with the lowest-level function, the one that does not make any calls to code that you are checking.
  - ✓ Then, move up the chain, checking each outer function in turn.
- You can do the same within a single function that you have split into logical sections.
  - ✓ Pick a section that you want to check and then figure out the inputs for it.
    - × In this case, the "inputs" consist of values for all variables that are used within the section of code you are walking through.
  - ✓ You should know which variables are relevant if you have determined the meaning of each variable.

Choose Inputs for Walkthroughs

- If the program has any state data that it keeps from one execution of the code to the next, think of possible values for that as well.
  - In object-oriented languages, the function that you look at might be a method on a class - in this case, the current state of the class member variables (the ones that are used in the function) is logically part of the inputs to that function.
- It should go without saying that when you select a test input, you need to know what your test output is supposed to be.
  - Otherwise, it makes it difficult to decipher whether the program works correctly.

Choose Inputs for Walkthroughs

- The steps are:
  - ✓ Code Coverage
  - ✓ Empty Input
  - ✓ Trivial Input
  - ✓ Already Solved Input
  - ✓ Error Input
  - ✓ Loops
  - ✓ Random Numbers

Code Coverage

- When designing inputs with the code in front of you, you have an advantage over others who are doing "black box" testing on the code - who can execute only the code and cannot see the source.
  - The advantage is that you can tailor your inputs to ensure that they exercise all the code.
- It might be tempting to think that any reasonably large or diverse group of inputs will naturally cover all the code.
  - In fact, code that is executed on every input is more likely to be correct than code that runs rarely - errors in the common code are more likely to have been found during initial development and debugging.

Code Coverage

- You cannot assume that all the code has been covered by your tests; instead, choose inputs that ensure it will be.
- One aspect of code that you must keep in mind is the "implied else," that is, everything that is done if an if() is true, is not done if the if() is false.
  - ✓ The most obvious case of an "implied else" is where no else body exists at all.
- In terms of choosing inputs, you have to cover the "implied else" also.

Empty Input

- Empty input is a situation where there is no data to work on.
  - ✓ Example: a program to sort an array is passed an array with zero elements; or a program to operate on strings is given an empty string.
- Typically, a program will handle this in one of two ways:
  - ✓ by explicitly checking for it at the beginning
  - ✓ by handling the empty case as part of the main algorithm
- Whichever way the code handles the empty case, you need to determine what an appropriate empty input would be, and walk through the code with that input.

Trivial Input

- Trivial input is the next step up from empty input:
  - A possible list of items turns out to have only one item, so the work to be done is trivial or nonexistent.
- As with empty input, trivial input might be handled by performing a special check at the beginning, often combined with a check for the empty case.
- Again, neither way is "right" or "wrong."
  - The goal is just to make sure the code works correctly when you walk through it with a trivial input.
  - Especially in cases where the trivial case is handled by the main algorithm, walking through the code - even if it manages to handle the trivial case correctly - can make you aware of a situation in which it would handle a nontrivial case incorrectly.

Already Solved Input

- Already solved input is for functions that are supposed to modify data in place.
  - ✓ It refers to a situation in which nothing needs to be modified.
- The already solved input exercises the code that determines if something needs to be done, without (hopefully) executing the code that actually does something.
- Unlike the empty and trivial inputs, it is usually impossible (or not worth the trouble) for the code to determine with an initial check whether the input is already solved.

Already Solved Input

- When designing input for the already solved case, one question is how long the input needs to be.
  - ✓ In general, using an input of between three and five "items" (where an item is one element in an array, one character in a string, and so on) is a good tradeoff between being short enough to feasibly walk through the code as it processes the entire input, and long enough to encounter any bugs that are dependent on the fact that a certain number of items are present in the input.
- Pay attention to cases where the code seems to be doing too much in processing the already solved case.
  - ✓ Moving data items around unnecessarily, even if they all wind up back in their original places, is certainly a performance issue, and might indicate a defect that will appear in some not already solved cases.

Error Input

- Error input is input that is just plain wrong.
- With error input, in addition to making sure that the function handles it without crashing, a walkthrough should verify that it behaves in the correct way.
- In many cases, an actual error input should be handled differently from, say, an empty input, by returning a specific error value or throwing an exception.

Error Input

- In other situations, where a function is nested within other code that is part of the same module, an error input might be considered an error on the part of the calling function, and by design should not be handled.
  - Of course, some functions do not have any input that could be considered a real error.
  - In most cases, it should be possible to come up with an error input and walk through it.

Random Numbers

- Some functions use a randomly generated number in their computations.
  - These functions typically use a random-number package written by someone else, either part of the language, the operating system, or a separate library.
- The main thing to worry about with random numbers is to check the exact range that the random number returns.
  - Some numbers return a value that is between 0 and a specified number; others, between 0 and 1.
  - In some cases, the top range of the random number is just less than the specified number, so they will never be equal.

Random Numbers

- The value returned by the random-number generator is another input to the code, even if it appears suddenly in the middle.
  - ✓ As such, you have to pick values for the random-number generator to return during your walkthrough.
- It's best to first pick those that are at the lower and upper limits; in the case just shown, those would be 0 and a number just below 1.
  - ✓ Picking other inputs usually depends on what is done next with the random number.
  - ✓ If the code does one of three things based on the result of the random number, pick three values to correspond to the three choices - it's likely that the values 0 and "just below 1" already covered two of the choices.

Random Numbers

- For random numbers that are used in a calculation as opposed to an explicit choice, picking a third choice that is halfway between the lower and upper limits is usually adequate.

Walk Through Each Section

- It can be difficult, especially after reading through lots of code, to avoid simply sliding over statements that look reasonable.
- The computer devotes its full attention to each statement as it is being executed, and you need to do the same.
  - You have to force yourself to focus on what is actually in the code, not what is supposed to be there or what you think is there - no matter if:
    - a statement seems obvious,
    - a constant definition looks trivial,
    - an expression seems correct at first glance.
- The steps are:
  - Track Variables
  - Code Layout
  - Loops

Track Variables

- When walking through code, you need to keep track of what value is in every variable, unless you have determined that a variable is no longer important to the function (and even then, you might discover that such a determination was false).
- Two ways to keep track of variables:
  - ✓ Say to yourself, as you begin to look at each statement - this can work well for simple cases.
  - ✓ Write down all the variables on a piece of paper - this way is better if there are many variables, or the statements contain complicated expressions where parsing would require too much brainpower for you to simultaneously remember what values were stored in every variable.

Track Variables

- Keep in mind that for every variable, every statement in the program either modifies the variable or does not modify the variable.
  - The computer never loses track and forgets to modify a variable if instructed to do so - writing it all down on paper helps prevent you from losing track.
    - It also helps you realize which variables change during the section and which ones remain constant.
- If you discover that the inputs you have selected make it too difficult to keep track of all the variables you can go back and change your inputs.
  - Keep in mind, however, that certain bugs might appear only with large enough inputs.

Code Layout

- The layout of the code in most languages is intended as a hint to a person who is reading the code
  - ✓ It usually is not used by the compiler or interpreter when determining how to execute a program.
- Unless a language specifically requires it, indentation and the placement of curly braces should not be used to infer the semantics of code
  - ✓ You have to check that the actual semantics are correct.
- On the other hand, in some languages, layout issues, such as indentation or which column a character appears in, are significant, and can cause the opposite sort of confusion, where you miss the significance of indentation.

Code Layout

- Improperly terminated comments can also obscure the true nature of code.
  - If you are debugging code and you have narrowed the problem down to a small section of code, but you simply cannot determine where the bug is, some languages allow you to remove the comments to check whether the defect is related to a statement unexpectedly being commented out.
- Be careful when reading complicated arithmetic expressions, especially those that do not use parentheses to make the order of evaluation explicit.
  - If you are not sure how an expression will be parsed, you can add parentheses yourself in a way that you feel is correct, and then see if this changes the program's behavior.

Loops

- Loops can be especially tricky to walk through because you cannot usually simulate every iteration of a loop.
  - ✓ With code that proceeds linearly without loops, it is often easy to spot defects by examining each line in turn.
- With any loop, pay attention to where the loop exits and where it exits to.
  - ✓ Normally, a loop exits at the end when the termination condition becomes false, but loops can also exit because of ***break*** statements in the middle, or ***return*** statements from inside a function.
    - × Tthe exit condition of a loop is only implicitly tested at the end of a loop.
  - ✓ If a loop has a break statement and where it will jump to.
  - ✓ Some languages have a way to specify code that is always executed when the loop ends.

Loops

- When the loop is done, it is important in those cases to be aware of what state a particular language will leave a loop counter in.
  - In particular, will it be set to the value it had during the last iteration, or one more than that?
- When you have a loop that needs to iterate many times, you have to choose certain iterations of the loop to walk through.
  - A good choice to begin with is to walk through the first iteration, the second iteration, the second-to-last iteration, and the last iteration.
- In cases where the result of an iteration depends on what happened in the previous iteration, you can often use an inductive process to prove to yourself that the loop is correct.
  - Assume the loop worked correctly on the previous iteration, and then see if this implies that it will work correctly on this one.

Questions?

V&V and UML Models

V&V and Models

- The quality of the model itself is one singularly important factor that influences the value of a model:
  - ✓ If the abstraction is incorrect, then obviously the reality eventually created out of that abstraction is likely to be incorrect.
  - ✓ An incorrect abstraction will also not reflect or represent the reality truthfully.
  - ✓ It is of immense importance in eventually driving quality benefits.

V&V and Models

- Modeling is limited by the following caveats:
  - ✓ A model, by its very nature, is an abstraction of the reality.
  - ✓ Unless a model is dynamic, it does not provide the correct sense of timing.
  - ✓ A model may be created for a specific situation or to handle a particular problem.
  - ✓ A model is a singular representation of possible multiple elements in reality.
  - ✓ The user of the model should be aware of the notations and language used to express the model.
  - ✓ Modeling casually, or at random, without due care and consideration for the nature of the models themselves, usually results in confusion in projects and can reduce productivity.
  - ✓ Models must change with the changing reality.
  - ✓ Processes play a significant role in steering modeling activities.

Model's Quality

- It is not the only aspect of quality in a project.
  - ✓ It exists within the context of other quality dimensions or levels, and these influence each other as well as model quality.
- Levels of quality:
  - ✓ Data quality
  - ✓ Code quality
  - ✓ Model quality
  - ✓ Architecture quality
  - ✓ Process quality
  - ✓ Management quality
  - ✓ Quality environment

The accuracy and reliability of the data, resulting in
The correctness of the programs and their
The correctness and completeness of the software
The quality of the system in terms of its ability to
The activities, tasks, roles and deliverables employed
Planning, budgeting and monitoring, as well as the
All aspects of creating and maintaining the quality of a project, including all of the above aspects of quality.

Model's Quality

- The aforementioned quality levels play a significant role in enhancing the overall quality of the output of a software project.
  - ✓ Most literature on quality, focuses on code and data quality.
  - ✓ Even when modeling appears in the discussion of quality, it is with the aim of creating good-quality software (data and algorithms).
  - ✓ In software projects without substantial modeling, code remains the primary output of the developers.
- In projects, code emerges from the developer's brain - directly.
  - ✓ The history of software development indicates has had disastrous effect on software projects.

Model's Quality

- Modeling is used not only to create the software solution but also to understand the problem.
  - ✓ As a result, modeling occurs in the problem, solution and background (architectural) spaces.
- The modeling output in such software projects transcends both data and code and results in a suite of visual models or diagrams.
  - ✓ While these models go on to improve the quality of the code produced, it is not just their influence on the implemented code that interests us but also their own quality - that is, the quality of the models themselves.
- Model quality is all about V&V of the models themselves.
  - ✓ The result is not only improved model quality, but also improved communication among project team members and among projects.

UML in modeling

- UML is not a methodology, but rather a common and standard set of notations and diagrams.
  - ✓ These are used by processes in varying ways to create the required models in the problem, solution and background modeling spaces.
- Large range of projects
  - ✓ New development projects
  - ✓ Integration projects
  - ✓ Package implementation
  - ✓ Outsourcing projects
  - ✓ Data warehousing and conversion projects
  - ✓ Educational projects

UML and quality

- UML has four main purposes:
  - ✓ Visualization
  - ✓ Specification
  - ✓ Construction
  - ✓ Documentation

UML and quality

- Visualization
  - ✓ UML notations and diagrams provide an excellent industry standard mechanism to represent pictorially the requirements, solution and architecture.
  - ✓ UML's ability to show business processes and software elements visually, spanning the entire life cycle of software development, provides the basis for extensive modeling in software development.
  - ✓ Its class representations, can bring the reality (real customers, accounts and transactions in a typical banking system) close to the people working in the solution space.
  - ✓ This quality of visualization is enhanced not only by the use of UML as a standard, but also because of the large number of CASE tools supporting these visual diagramming techniques.

UML and quality

- Specification
  - Together with visual representations, UML facilitates the specification of some of its artifacts.
  - UML specifications help enhance the quality of modeling:
    - they enable additional descriptions of the visual models
    - they enable members of a project team to decide which areas of a particular diagram or element they want to specify
    - they allow them (through CASE tools) to make the specifications available to all stakeholders.
    - The specifications can be made available in various formats: a company's intranet Web page, a set of Word documents or a report.

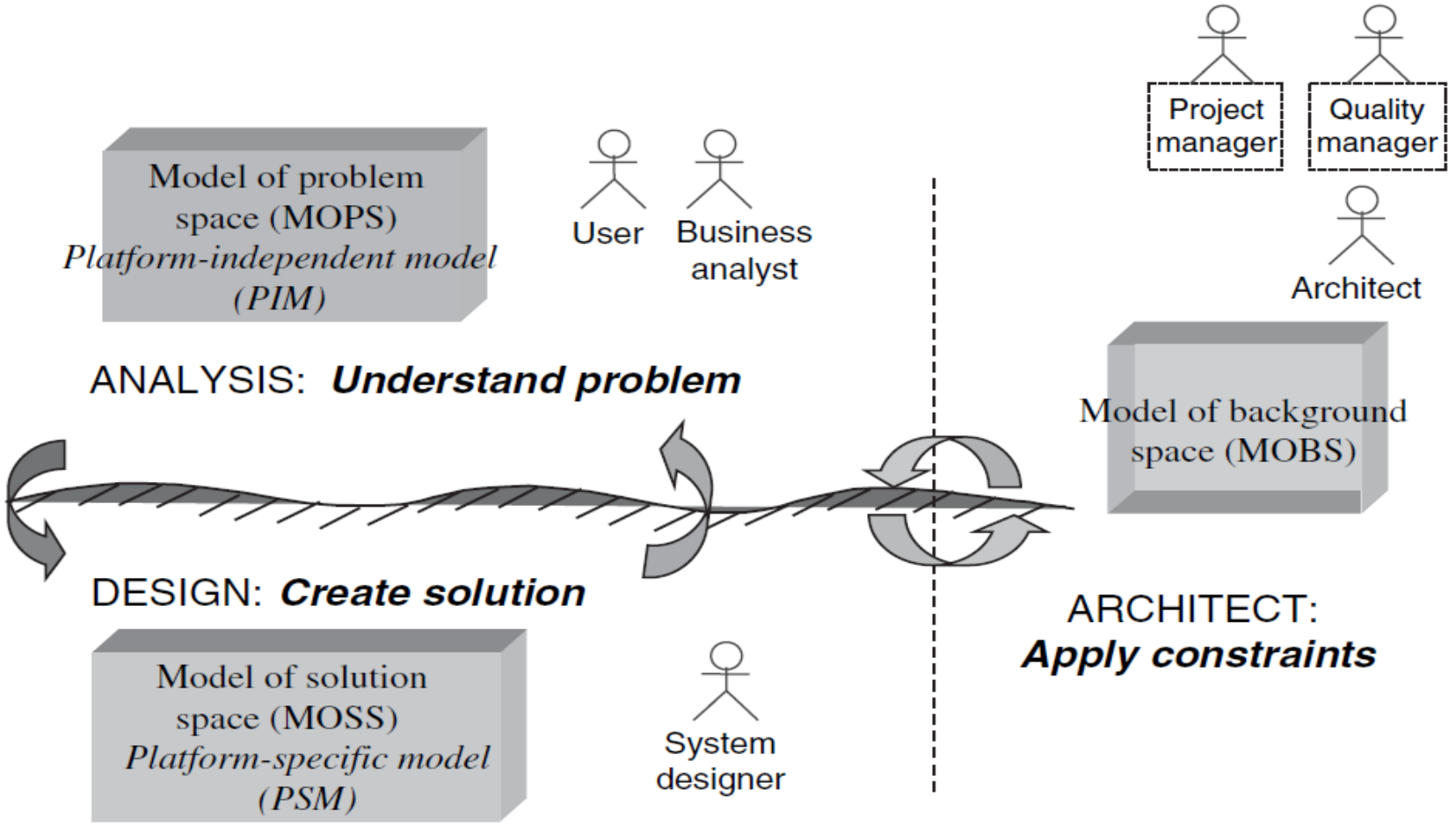
UML and quality

- Construction
  - ✓ UML can also be used for software construction, as it is possible to generate code from UML visual representations.
- A piece of software that is constructed based on formal UML-based modeling is likely to fare much better during its own V&V.
  - ✓ Classes and class diagrams, together with their specifications (e.g., accessibility options, relationships, multiplicities), ensure that the code generated through these models is correctly produced and is inherently superior to hand-crafted code (i.e., code without models).

UML and quality

- Documentation
  - With the help of UML, additional and detailed documentation can be provided to enhance the aforementioned specifications and visual representations.
    - Documentation has become paramount - not only the type that accompanies the code, but also the type that goes with models, prototypes and other such artifacts.
  - In UML, diagrams have corresponding documentation, which may be separate from the formal specifications and which goes a long way toward explaining the intricacies of visual models.

UML and modeling spaces



Modeling spaces & UML diagrams

- These diagrams have a set of underlying rules that specify how to create them.
  - The rigor of these rules is encapsulated in what is known as the OMG's "meta-model".
    - The meta-model also helps to provide rules for cross-diagram dependencies.
  - The importance of the meta-model is that it renders the UML elastic
    - It can be stretched or shrunk, depending on the needs of the project.
    - The elasticity of UML, the extent and depth to which the UML diagrams are applied in creating models are crucial to the success of projects using UML.
  - Not all diagrams apply to all situations.
  - Not all diagrams are relevant to a particular role within a project.

MOPS & UML diagrams

- The problem space will need the UML diagrams that help the modeler understand the problem without going into technological detail.
  - ✓ As a nontechnical description of what is happening with the user or the business
- The UML diagrams that help express what is expected of the system, rather than how the system will be implemented, are of interest here.

MOPS & UML diagrams

- Use case diagrams
  - ✓ Provide the overall view and scope of functionality.
    - × The use cases within these diagrams contain the behavioral (or functional) description of the system.
- Activity diagrams
  - ✓ Provide a pictorial representation of the flow anywhere in MOPS.
    - × In MOPS, these diagrams work more or less like flowcharts, depicting the flow within the use cases or even showing the dependencies among various use cases.
- Sequence and State machine diagrams
  - ✓ Occasionally used to help us understand the dynamicity and behavior of the problem better.

MOPS & UML diagrams

- Class diagrams
  - ✓ Provide the structure of the domain model.
    - × In the problem space, these diagrams represent business domain entities (such as Account and Customer in a banking domain), not the details of their implementation in a programming language.
- Interaction overview diagrams
  - ✓ These provide an overview of the flow and/or dependencies between other diagrams.
- Package diagrams
  - ✓ Can be used in the problem space to organize and scope the requirements.
    - × Domain experts, who have a fairly good understanding not only of the current problem but also of the overall domain in which the problem exists, help provide a good understanding of the likely packages in the system.

MOSS & UML diagrams

- The solution space is primarily involved in the description of how the solution will be implemented.
  - ✓ This solution model requires extra knowledge and information about the facilities provided by the programming languages, corresponding databases, middleware, Web application solutions and a number of other technical areas.
- MOSS contains a solution-level design expressed by:
  - ✓ technical or lower-level class diagrams,
  - ✓ technical sequence diagrams,
  - ✓ detailed state machine diagrams representing events and transitions,
  - ✓ designs of individual classes and corresponding advance class diagrams.

MOSS & UML diagrams

- Because MOSS is a technical description of how to solve the problem, the UML diagrams within MOSS are also technical in nature.
  - ✓ Furthermore, even the diagrams drawn in MOPS are embellished in the solution space with additional technical details based on the programming languages and databases.
- The primary diagrams used in MOSS are the class diagrams together with their lowermost details, including:
  - ✓ attributes,
  - ✓ types of attributes,
  - ✓ their initial values,
  - ✓ signatures of the class operations (including their parameters and return values) and
  - ✓ so on.

MOSS & UML diagrams

- Class diagrams can be followed by diagrams like the sequence diagrams together with their detailed signatures, message types, return protocols and so on.
- Modelers may also use the communication diagrams for the same purpose as sequence diagrams.
- Occasionally state machine diagrams can be used to provide the dynamic aspect of the life cycle of a complex or very important object.
- Recently introduced timing diagrams show state changes to multiple objects at the same time.
- Composite structure diagrams depict the run-time structure of components and objects.

MOBS & UML diagrams

- MOBS incorporates two major aspects of software development that are not covered by MOPS or MOSS:
  - ✓ Management
  - ✓ Architecture
- MOBS is an architectural model in the background space that influences models in both problem and solution spaces through constraints.
- Of the two major aspects of work in the background space, management work relates primarily to planning.
  - ✓ Planning deals mainly with the entire project and does not necessarily form part of the problem or solution space.
  - ✓ Management work in the background space includes issues from both problem and solution spaces but is not part of either of them.

MOBS & UML diagrams

- Several aspects of planning are handled in the background by the project manager.
- These include planning the project:
  - ✓ resourcing the project's hardware, software, staff and other facilities;
  - ✓ budgeting and performing cost-benefit analysis;
  - ✓ tracking the project as it progresses through various iterations; and
  - ✓ providing checkpoints for various quality-related activities.
- Background space activities are related to management work.
  - ✓ UML is not a management modeling language.
  - ✓ UML does not provide direct notations and diagrams to document project plans and resources.
- The project planning aspect is best to deal with the process techniques as well as process tools.

MOBS & UML diagrams

- Architectural work deals with a large amount of technical background work.
  - ✓ This work includes consideration of patterns, reuse, platforms, Web application servers, middleware applications, operational requirements and so on.
- Background space also includes issues such as reuse of programs and system designs, as well as system and enterprise architecture.
  - ✓ Work in this space requires knowledge of the development as well as the operational environment of the organization, availability of reusable architecture and designs, and how they might fit together in MOPS and MOSS.

MOBS & UML diagrams

- The UML diagrams of interest in the background space are the ones that help us create a good system architecture that strives to achieve all the good things of object orientation.
  - ✓ Reusability, patterns and middleware need to be expressed correctly in MOBS, and UML provides the means to do so.
- Users should expect
  - ✓ a large amount of strategic technical work in the background space that will consider the architecture of the current solution
  - ✓ the existing architecture of the organization's technical environment
  - ✓ the operational requirements of the system
  - ✓ the needs of the system in terms of its stress, volume and bandwidth.

MOBS & UML diagrams

- It is necessary to relate the solution-level classes closely to the component diagrams drawn in the background space.
  - ✓ These component diagrams contain the .EXEs and .DLLs and are closely associated with the solution level class diagrams, providing the final steps in the system development exercise before the user runs the application.
  - ✓ When run-time components are modeled, they result in composite structure diagrams, which may also be used in the background space to specify and discuss the architecture of a component or a class.
- Operational issues are being expressed properly using UML diagrams in the background space.
  - ✓ UML provides help and support in modeling the operational environment (or deployment environment) of the system by means of deployment diagrams.

MOBS & UML diagrams

- A combination of component and deployment diagrams can provide a basis for discussions between the architects and designers of the system concerning where and how the components will reside and execute.
  - ✓ Using the extension mechanisms of UML, one can develop diagrams that help express Web application architectures
- These background UML diagrams also have a positive effect on architecting quality by providing standard means of mapping designs to existing and proven architectures.

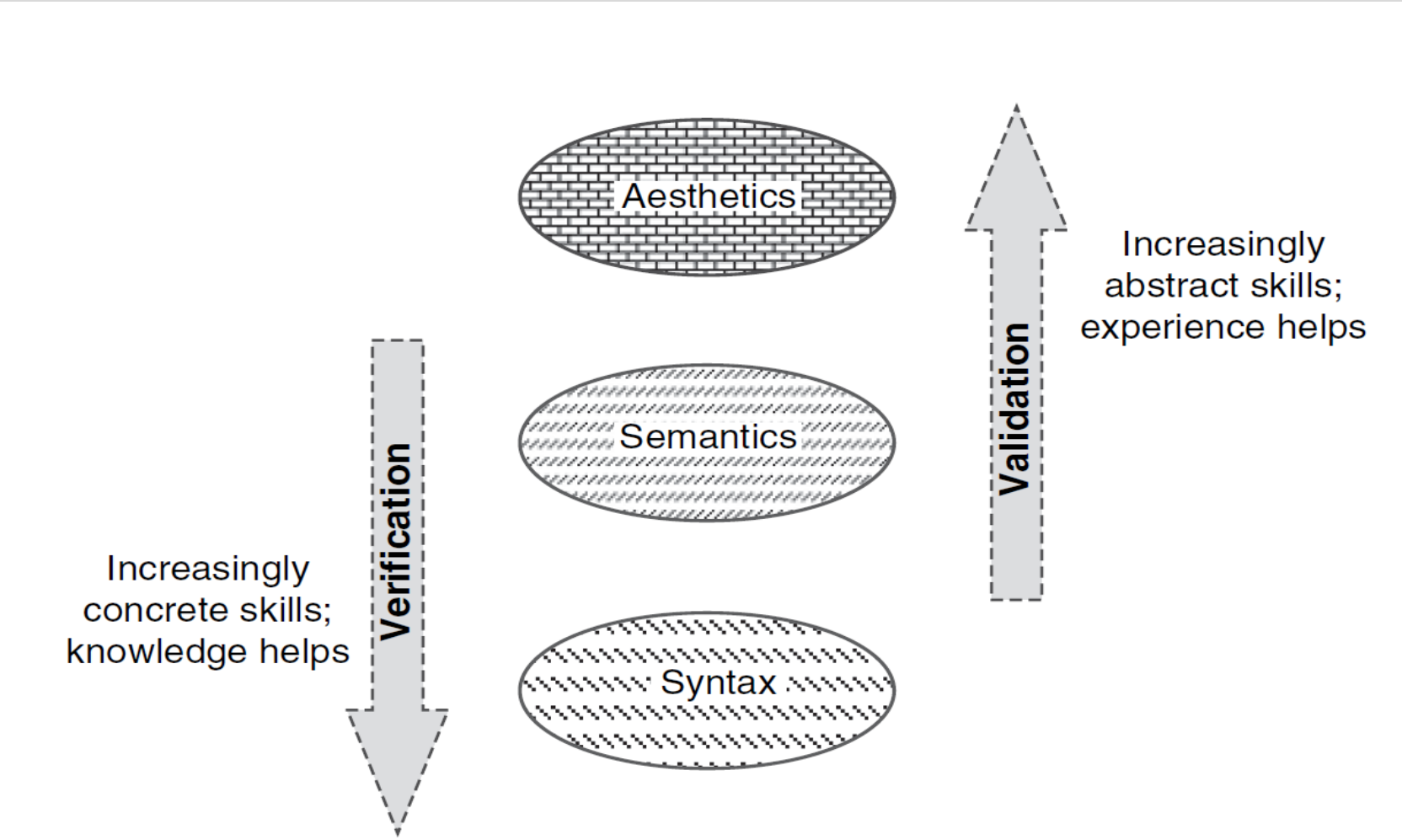
Modeling spaces & UML diagrams

UML Diagrams	MOPS	MOSS	MOBS
Use case	*****	**	*
Activity	*****	**	*
Class	***	*****	**
Sequence	****	*****	*
Interaction overview	****	**	**
Communication	*	***	*
Object	*	*****	***
State machine	***	****	**
Composite structure	*	*****	****
Component	*	***	*****
Deployment	**	**	*****
Package	***	**	****
Timing	*	***	***

V&V and models' quality

- The major part of V&V deals with the visual aspects of the model.
  - ✓ This can lead not only to detection of errors in the model - quality checks that ensure validation of the model.
  - ✓ Appropriate quality assurance and process-related activities aimed at the prevention of errors.
- For V&V of a software artifact, there are three levels of checks: syntax, semantics and aesthetics.
  - ✓ The words "syntax," "semantics" and "aesthetics" are chosen to reflect the techniques or means of accomplishing the V&V of the models.

V&V and models' quality



V&V and models' quality

- Syntax, semantics and aesthetics have close parallels to the quality approach, who created a framework with three axes for quality assessment:
  - ✓ Language
  - ✓ Domain
  - ✓ Pragmatics
- These quality assessment are translated axes into:
  - ✓ Syntactic quality
  - ✓ Semantic quality
  - ✓ Pragmatic quality
- New axis providing the theoretical background on which the current quality checks are built.

V&V and models' quality

- V&V of software models:
  - ✓ All quality models should be syntactically correct, thereby adhering to the rules of the UML (as a modeling language) they are meant to follow.
  - ✓ All quality models should represent their intended semantic meanings and should do so consistently.
  - ✓ All quality models should have good aesthetics, demonstrating the creativity and farsightedness of their modelers.
    - This means that software models should be symmetric, complete and pleasing in what they represent.

Quality Models: Syntax

- All languages have a syntax.
- Two major characteristics of UML differentiate it from the other languages:
  - UML is a visual language, which means that it has a substantial amount of notation and many diagram specifications.
  - UML is a modeling language, which means that it is not intended primarily to be compiled and used in production of code (as programming languages are).
    - Although the trend toward support for both “action semantics” in UML 2.0 and in MDA will likely involve the use of UML in this context in the future.

Quality Models: Syntax

- Incorrect syntax affects the quality of visualization and specification.
  - ✓ Incorrect syntax at the diagram level percolates down to the construction level, causing errors in creating the software code.
- In UML-based models, when we apply syntax checks, we ensure that each of the diagrams that make up the model has been created in conformance with the standards and ‘good practice’ guidelines.
  - ✓ The notations used, the diagram extensions annotated and the corresponding explanations on the diagrams all follow the syntax standard of the modeling language.
- CASE tools are helpful to ensure that syntax errors are kept to a minimum.

Quality Models: Syntax

- Permissible variations on these diagrams in complying with the meta-model can become a project-specific part of the syntax checks.
- Syntactic correctness greatly enhances the readability of diagrams, especially when these diagrams have to be read by different groups in different organizations in several countries
  - ✓ This is a typical software outsourcing scenario.

Quality Models: Semantics

- A model, although syntactically correct, would fail to achieve the all-important semantic correctness.
  - ✓ The semantic aspect of model quality ensures not only that the diagrams produced are correct, but also that they faithfully represent the underlying reality represented in the domain.
- Once again, models in general are not executable
  - ✓ It is not possible to verify and validate their purpose by simply "executing" them, as one would the final software product (the executable).
  - ✓ Alternative evaluation techniques need to be used.
    - ✗ The traditional and well-known quality techniques of walkthroughs and inspections are extremely valuable.

Quality Models: Aesthetics

- Very simply: aesthetics implies style.
  - ✓ Although the code (or, for that matter, any other deliverable) may be accurate (syntactically) and meaningful (semantically), difference still arises due to its style.
  - ✓ The style of modeling has a bearing on the models' readability, comprehensibility and so on.
- The aesthetic size consideration is studied in terms of the granularity of the UML models and requires a good metrics program within the organization to enable it to improve the aesthetics of the model.
  - ✓ A model offer a high level of customer satisfaction, primarily to the members of the design team but also in their discussions with the business endusers.

V&V Checks

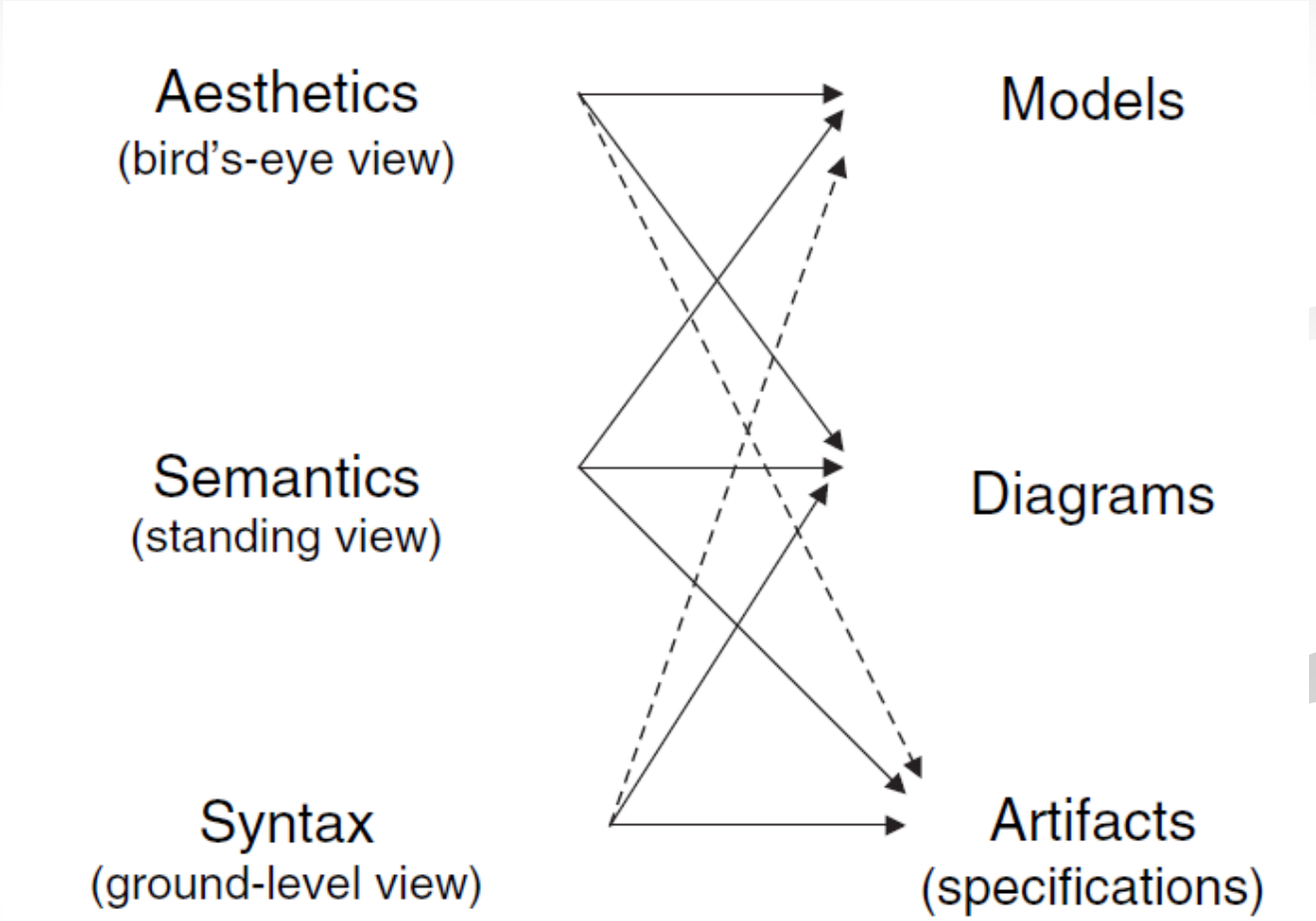
- The V&V checks of syntax, semantics and aesthetics:
  - Walkthroughs - may be performed individually, and help weed out syntax errors (more than semantic errors).
  - Inspections - are more rigorous than walkthroughs, are usually carried out by another person or party, and can identify both syntax and semantic errors.
  - Reviews - increase in formality and focus on working in a group to identify errors.
    - The syntax checks are less important during reviews, but the semantics and aesthetics start becoming important.
  - Audits - formal and possibly external to the project and even the organization.
    - They are not very helpful at the syntax level, but they are extremely valuable in carrying out aesthetic checks of the entire model.

Quality checks of UML diagrams

- The three levels as follows:
  - ✓ The individual elements, or “things” that make up the diagrams (a “ground-level view” of the model).
  - ✓ The UML diagrams and the validity of their syntax, semantics and aesthetics (a “standing view” of the model).
  - ✓ A combination of interdependent diagrams called a “model” (a “bird’s-eye view” of the model).
- It is not necessary to apply all types of checks to all of the artifacts, diagrams and models produced.

Quality checks of UML diagrams

- The three levels as follows:



Correctness (syntax checks)

- We must to expand our V&V effort to levels beyond just one diagram.
- In syntax checks, we are looking at the ground-level view of the models.
  - ✓ This includes the artifacts and elements of UML, as well as their specifications and documentation.
- When we check the syntax of UML elements, we focus primarily on the correctness of representation as mandated by UML.
  - ✓ Therefore, during syntax checks the semantics, or the meaning behind the notations and diagrams, are not the focus of checking.

Correctness (syntax checks)

- Syntax checks:
  - ✓ In a use case diagram, basic syntax checks apply to the artifacts or elements that make up the diagram, such as the actors and the use cases.
  - ✓ In a class diagram, basic syntax checks apply to a class first and whatever is represented within the class.
- This syntax check for an element or artifact is followed by a check of the validity of the diagram itself.
  - ✓ Instead of focusing on one element at this level, we inspect the entire diagram and ensure that it is syntactically correct.
- If these syntax checks for the elements and the diagrams that comprise them are conducted correctly, they ensure the correctness of the UML diagrams.

Correctness (syntax checks)

- Example: a class diagram that contains *Car* as a class.
  - The syntax check of the correctness of this artifact would be something like this:
    - Is the *Car* class represented correctly by attributes and operations?
    - Do the attributes have correct types and do the operations have correct signatures?
    - Is the *Car* class properly divided into three compartments?
    - Is the *Car* class compilable ?
      - This syntax check will apply in the solution space.

Completeness and Consistency (semantics checks)

- This check focuses not on the correctness of representation but on the completeness of the meaning behind the notation.
  - ✓ Example: “Does the *Car* class as named in this model actually represent a car or does it represent a garbage bin?”
- Semantic checks are best performed from a standing-level view of the UML models.
  - ✓ The meaning of one element of UML depends on many other elements and on the context in which it is used.
  - ✓ We move away from the ground-level check of the correctness of representation and focus on the purpose of representation.

Completeness and Consistency (semantics checks)

- Semantic checks become more intense at the diagram level rather than just at an element level.
  - ✓ It is not just one element on the diagram but rather the entire diagram that becomes visible and important.
- Semantic checks also deal with consistency between diagrams, which includes dependencies between elements.
  - ✓ Will need a detailed diagram-level semantic check.
  - ✓ This check will also include many crossdiagram dependency checks that extend the semantic check to more than one diagram.
- Semantic checks apply to each of the UML diagrams intensely, as well as to the entire model.

Completeness and Consistency (semantics checks)

- The ‘Car’ Example:
  - ✓ Semantic checks also deal with consistency between diagrams, which includes dependencies between *Door* and *Engine* and between *Wheel* and *Steering*.
  - ✓ While a class *Door* may have been correctly represented (syntactically correct) and may mean a *Door* (semantically correct), the dependencies between *Door* and car, or between door and driver (or even between door and burglar), will need a detailed diagram-level semantic check.
  - ✓ Semantic checks also focus on whether this class is given a unique and coherent set of attributes and responsibilities to handle or whether it is made to handle more responsibilities than just Car.
    - ✗ Do the Driver-related operations also appear in Car?

Symmetry and Consistency (aesthetics checks)

- The aesthetic checks of diagrams and models add a different dimension to the quality assurance activities.
- They deal not with correctness or completeness but rather with the overall consistency and symmetry of the UML diagrams and models.
  - ✓ Not just one diagram, but many diagrams, their interrelationships, and their look and feel
- They also require some knowledge and understanding of the process being followed in the creation of the models and the software.
  - ✓ The process ensures that the they are applied to the entire model rather than to one element or diagram.

Symmetry and Consistency (aesthetics checks)

- Aesthetic checks look at the entire model (MOPS, MOSS, MOBS or any other) to determine whether or not it is symmetric and in balance.
  - ✓ If a class diagram in a model has too many classes, aesthetic checks will ensure redistribution of classes.
- A good understanding of the aesthetic checks results in diagrams and models that do not look ugly, irrespective of their correctness.

Symmetry and Consistency (aesthetics checks)

- The 'Car' Example:
  - The aesthetic checks of the *Car* class involve checking the dependency of *Car* on other classes and their relationships with persistent and graphical user interface (GUI) class cross-functional dependencies.
    - This requires cross-checks between various UML diagrams that contain the *Car* as well as checks of their consistency.
  - Aesthetic checks focus on whether the *Car* has too many or too few attributes and responsibilities.

The result

- Together, the quality checks ensure that the artifacts we produce in UML, the diagrams that represent what should be happening in the system, and the models that contain diagrams and their detailed corresponding documentation are all correct, complete and consistent.

Questions?

V&V and UML Models

Strengths & Weaknesses of UML diagrams

Strengths & Weaknesses

- UML has an expansive suite of diagrams that can help modeling in projects of varying types and sizes.
- Characteristics of the UML diagrams are divided into two groups:
  - ✓ Intrinsic characteristics
    - × They exhibit irrespective of the type or size of the project in which they are used.
    - × They are both strengths and weaknesses of UML diagrams.
  - ✓ Extrinsic characteristics
    - × They become important when these diagrams are applied in creating practical models.
    - × Are dependent not only on the modeling spaces in which the diagrams are used, but also on the type of project in which they are applied.
    - × They are derived from the particular objectives of the project in which the diagrams are used.

SWOT analysis

- Strengths
  - ✓ Intrinsic strengths of the diagram represented by the reason for having that diagram in UML.
  - ✓ The strength of the diagram remains the same irrespective of the modeling space, roles of people, and type and size of the project.
- Weaknesses
  - ✓ Intrinsic weaknesses of the diagram that are due primarily to the lack of modeling capabilities of that diagram, irrespective of the modeling space, roles of people, and type and size of the project.

SWOT analysis

- Objectives
  - ✓ Extrinsic usage or purposes of the diagram when applied in a particular modeling space.
  - ✓ The objectives can also change, depending on the project size and type.
    - × Objectives help narrow the focus of the UML diagrams in application, thereby capitalizing on their practical value in the context of a project.
- Traps
  - ✓ Problems faced by practitioners when they start using the UML diagrams in projects.
    - × They also vary, depending on the project characteristics.
  - ✓ The traps in using the diagrams usually become apparent after the modelers have tried them out in projects.
    - × Upfront knowledge of these traps is very helpful in focusing the project team's effort to improve the quality of these diagrams.

SWOT analysis

Intrinsic	**Strengths** Strengths represent the intrinsic (inherent) positive characteristics of the diagrams irrespective of where they are used.	**Weaknesses** Weaknesses represent the intrinsic (inherent) negative characteristics of the diagram irrespective of where they are used.
Extrinsic	**Objectives** Represent the purpose of the diagram in practical modeling in the modeling spaces.	**Traps** Represent the problems/ traps encountered in usage of the diagram in the modeling spaces.

V&V of the MOPS

- UML in the problem space is used to communicate events in the business arena.
  - ✓ Therefore, in creating MOPS, we consider only those elements and diagrams of UML that are helpful in understanding the business and functional requirements of the project.
  - ✓ In the problem space, the purpose of UML is to enable understanding and documentation of requirements.
- The analysis outlines a checklist-based strategy that can be used for V&V of the UML diagrams in MOPS.
  - ✓ Diagrams: use case, activity, class, sequence, package and interaction overview diagrams.
  - ✓ V&V includes syntax, semantics and Aesthetic checks of diagrams.

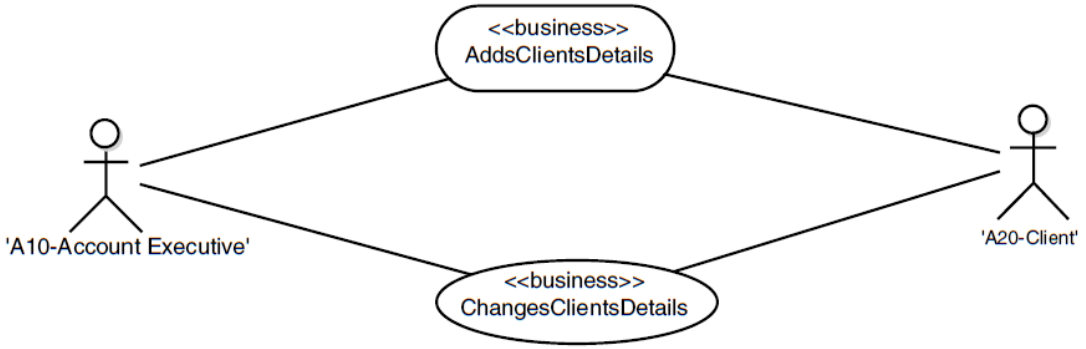
Problem Space

V&V of the MOPS

- It is important to note that the diagrams are independent of implementation considerations.
- All technological issues dealing with how to implement a solution are deferred until the solution space is modeled.
  - ✓ Example: Potential classes are identifying as entities in MOPS, whereas these classes are converted to language-specific implementation classes in MOSS.

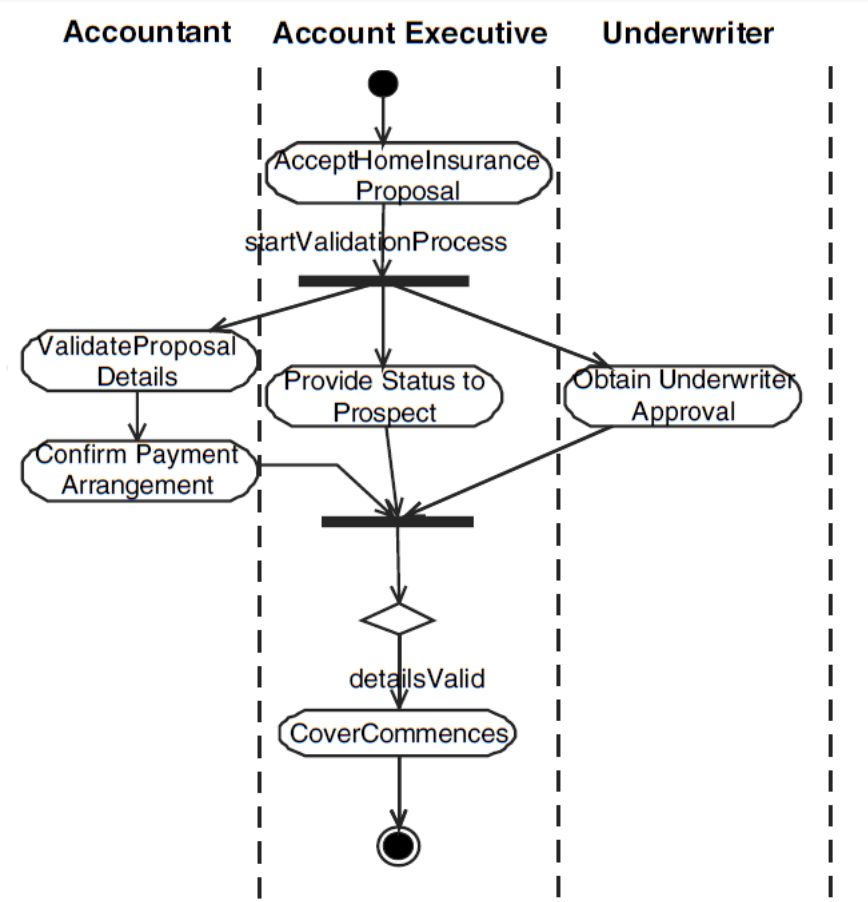
V&V of the MOPS

- Use case diagrams
  - ✓ Used as a primary means of interacting with users and understanding the problem.
  - ✓ Greater importance is the specification of the use cases themselves, which describes the interactions between the user and the system.



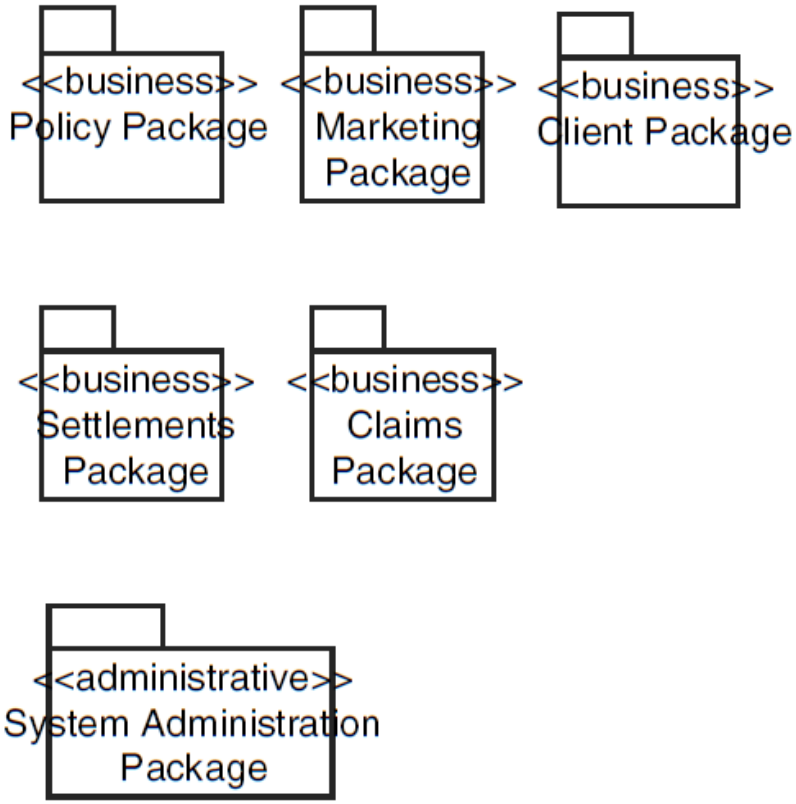
V&V of the MOPS

- Activity diagrams
  - ✓ Used primarily to understand, in further detail, the flow in the use case.
  - ✓ They can also be drawn to visualize the overall flow of the system.



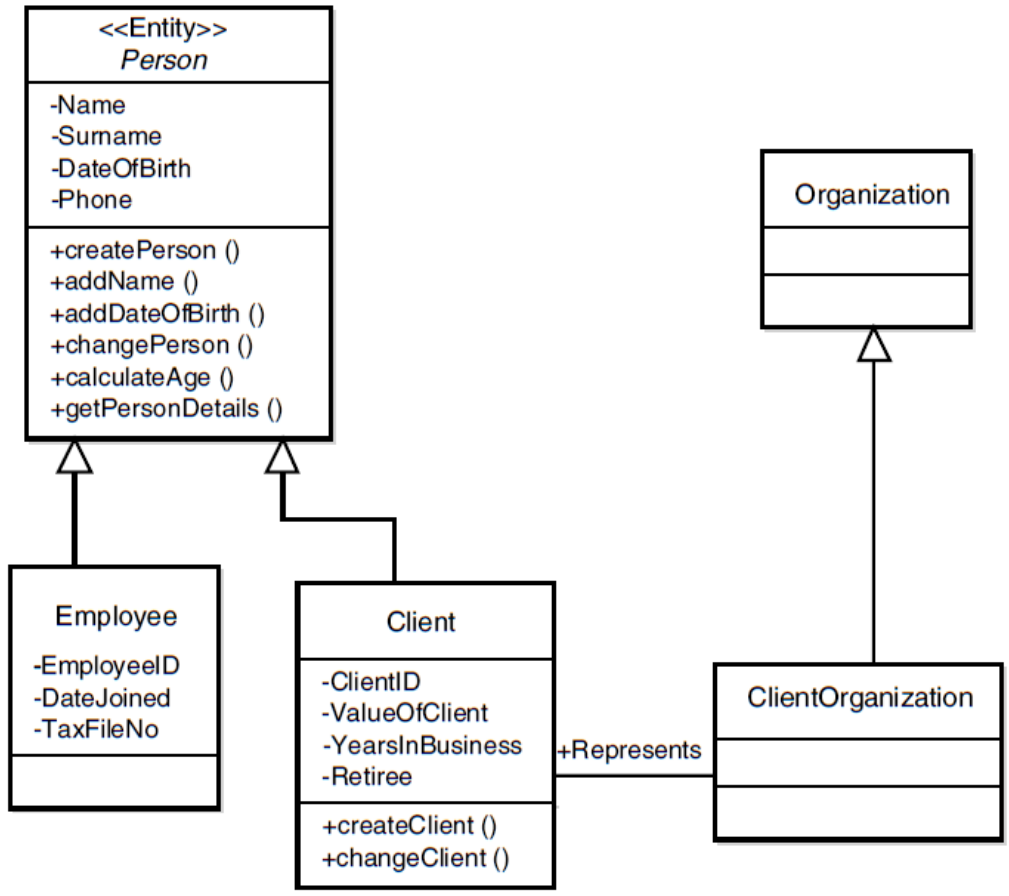
V&V of the MOPS

- Package diagrams
  - ✓ In MOPS, used as a “grouping mechanism” for the entire project, resulting in a well-organized project (project management).



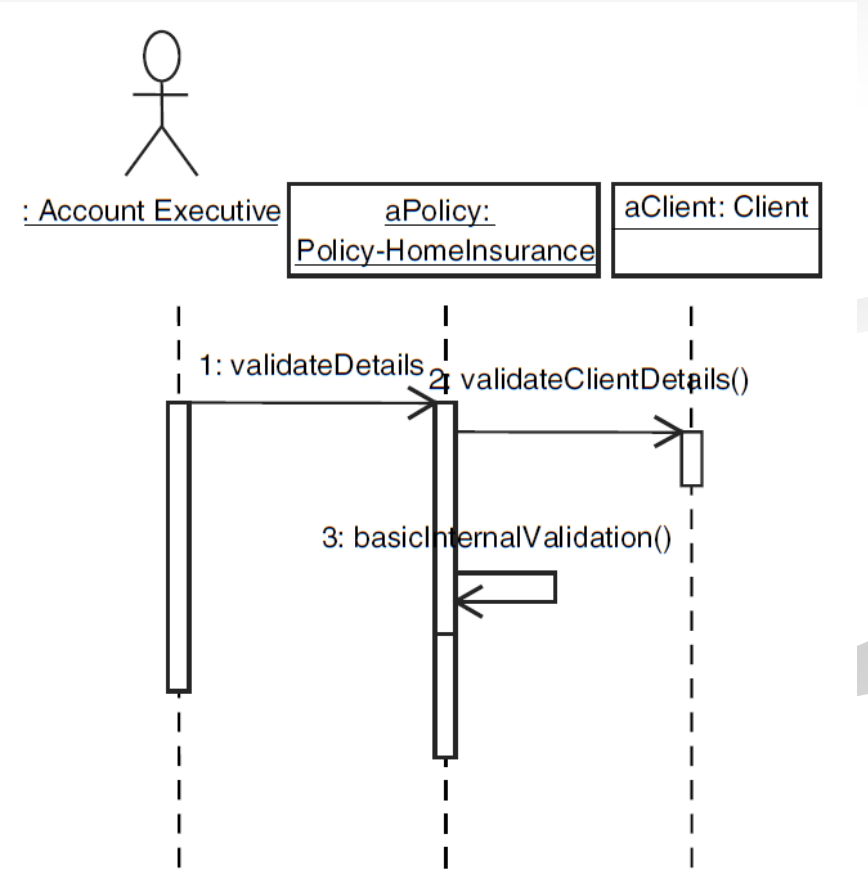
V&V of the MOPS

- Class diagrams
  - ✓ In MOPS, used to mod
    relationships (classes a
  - ✓ These class diagrams
    models" or "business c



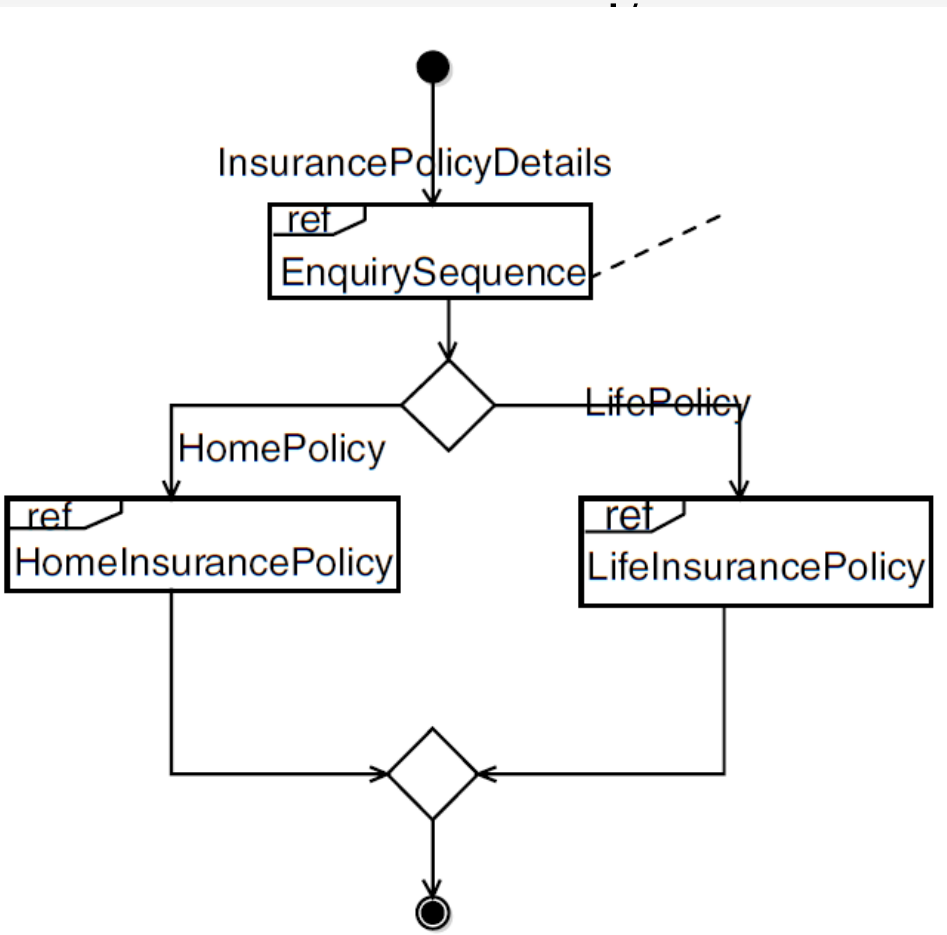
V&V of the MOPS

- Sequence diagrams
  - ✓ In MOPS, used to document complex and/or important scenarios that represent interactions within a use case, interactions among business objects or interactions described directly by business users (messages that become methods in a class).



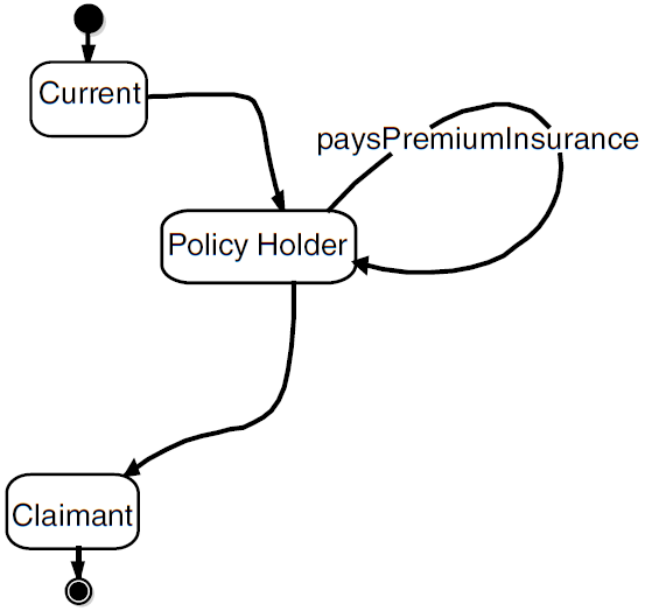
V&V of the MOPS

- Interaction overview diagrams
  - ✓ Used to model the flow b
    cases, with those sequer
    an element (reference) c
  - ✓ They can also show us h
    sending and receiving a
    distribute the workload to



V&V of the MOPS

- State machine diagrams
  - ✓ May optionally be used in MOPS to model the complex and/or important transitions undergone by a business object.
  - ✓ These transitions are the life cycle of described by the business users.
  - ✓ They are useful to model the life cycl mentioned above in the interaction o



V&V of the MOPS

- In addition, there are other important considerations in enhancing the quality of the model in the problem space.
  - ✓ These considerations are primarily those of aesthetics at the highest level of the model, as well as the semantic meanings of various cross-diagram dependencies.
- Cross-diagram dependencies are:
  - ✓ Completeness, correctness and consistency of diagrams.
  - ✓ Dependencies of the various diagrams on each other
  - ✓ Completeness of documentation outside the diagrams.
  - ✓ Iterative dependency on other models.
  - ✓ Third-party documentations

Use Case to Activity Diagram

- Have you considered an activity diagram for every use case?
  - ✓ This is not mandatory.
  - ✓ It is highly advisable
    - × Activity diagrams show pictorially what is happening within a use case.
- If an activity diagram is drawn for a use case, for every step within the text or flow of the use case, is there a corresponding activity within the activity diagram?
  - ✓ This check will influence (and update) both the activity diagram and the use case documentation.
- Some of these checks will happen as suggested by the process for software development. Other checks will have to be carried out outside of the process.
  - ✓ The quality techniques can be applied to these checks.

Use Case to Class Diagram

- It is normal and acceptable for multiple use cases to influence class diagrams.
  - ✓ It is important to consider additional classes that may not have appeared by simply analyzing the use cases.
- Going back and forth between use case descriptions and classes on a class diagram should be encouraged.
  - ✓ While use case diagrams and their descriptions provide the starting point for class identification, the reverse should also be carefully considered.
- It is worthwhile ensuring that every class can be traced to the descriptions of use cases and that
  - ✓ If there is no trace, there is a good reason for this lack of correlation between a class and at least one use case.

Class to Sequence Diagram

- This is one of the important cross-diagram dependencies in UML diagrams - sequences diagrams provide a pictorial representation of an example of a use case.
  - ✓ Therefore, by going through the sequence diagram, it is possible to prepare a list of potential classes.
  - ✓ Sequence diagrams are more precise than use case descriptions.
- They are also related closely to the class diagrams by the UML CASE tools.
  - ✓ A new object on a sequence diagram ⇨ A new class in a class diagram if that class does not exist.
  - ✓ A new message on a sequence diagram ⇨ A new method in a class on a class diagram if that method does not exist.
  - ✓ The message signatures ⇨ The method parameters on a class in a class diagram.

Interaction Overview Diagram to Sequence Diagram and Use Case

- Interaction overview diagrams provide an overview of the flow between sequences and use cases.
  - ✓ They reference the sequence diagrams and use cases in the use case diagrams.
- Ensure that the elements shown on the interaction overview diagrams have corresponding elements and/or diagrams in MOPS.

Quality of Documentation Associated with MOPS

- Some documentation may not fit within the UML specifications and documentation in the problem space:
  - ✓ business policies and procedures;
  - ✓ mathematical formulas;
  - ✓ government regulations.
- Business rules that may not fall within the purview of a single use case or class will also remain outside UML documentation.
- These additional types of documentation should be tied together and verified for correctness and completeness.

Aesthetics of MOPS

- It is worthwhile to consider the entire MOPS from a bird's-eye viewpoint.
  - ✓ After completing all the checks on the UML elements and diagrams and crosschecking the dependencies among the diagrams.
  - ✓ It is essential to take a step back, after a reasonable amount of modeling is done, to consider the aesthetic quality of the entire MOPS.
- It is important to perform this check at the highest level
  - ✓ It is not the check of a specific element or diagram in the model.
  - ✓ It is not likely to be as obvious as the other checks.
- The aesthetics of the MOPS should be checked at the end of initial, major and final iterations.

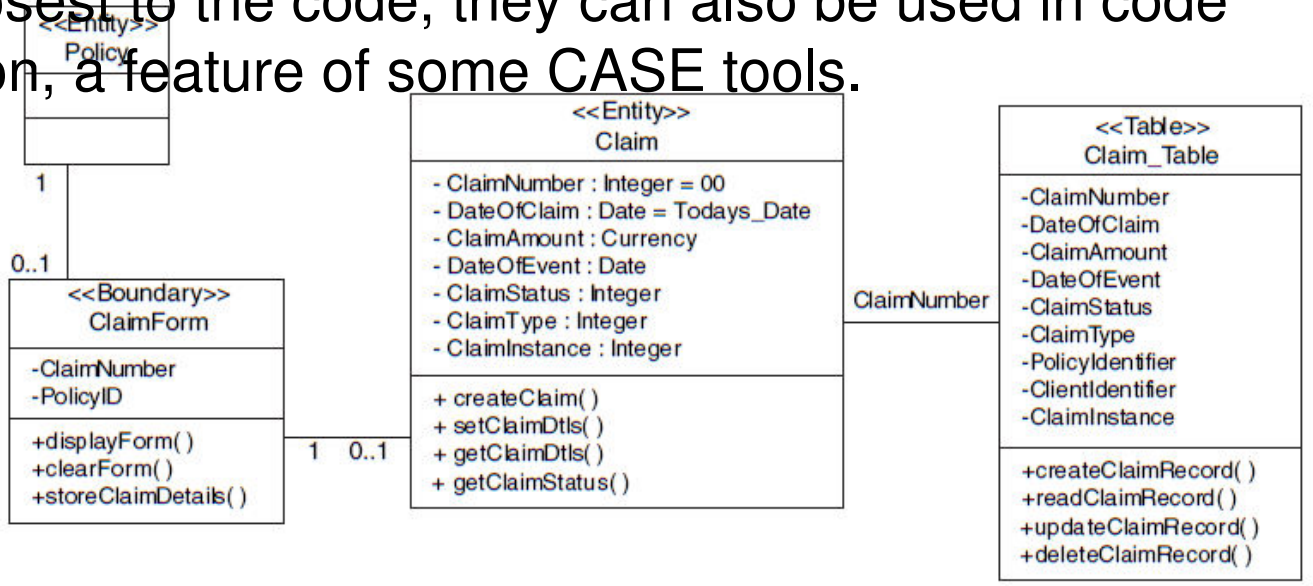
Solution Space

V&V of the MOSS

- UML diagrams in the solution space include:
  - ✓ advanced class diagrams;
  - ✓ advanced sequence diagrams;
  - ✓ corresponding communication diagrams;
  - ✓ object diagrams;
  - ✓ advanced state machine diagrams;
  - ✓ timing diagrams.
- The evaluation of each of these diagrams need detailed V&V checks of their syntactic correctness, semantic completeness and consistency and aesthetic quality.

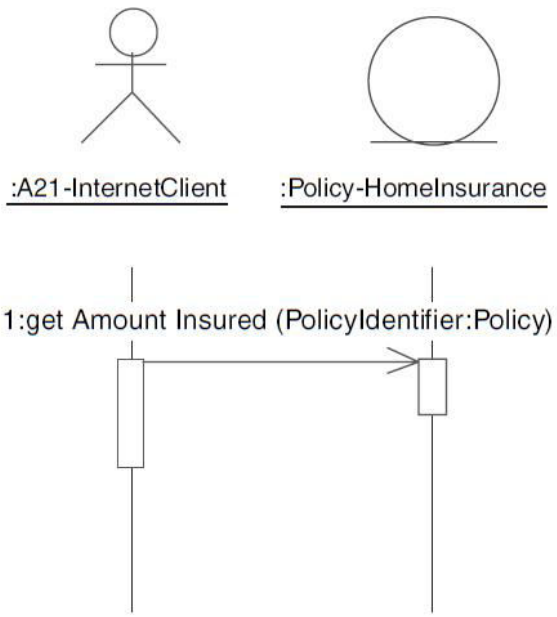
V&V of the MOSS

- Class diagrams (advanced)
  - ✓ The most important diagrams in the solution space.
  - ✓ They embellish the business class diagrams to make them implementable.
    - ✘ Being closest to the code, they can also be used in code generation, a feature of some CASE tools.



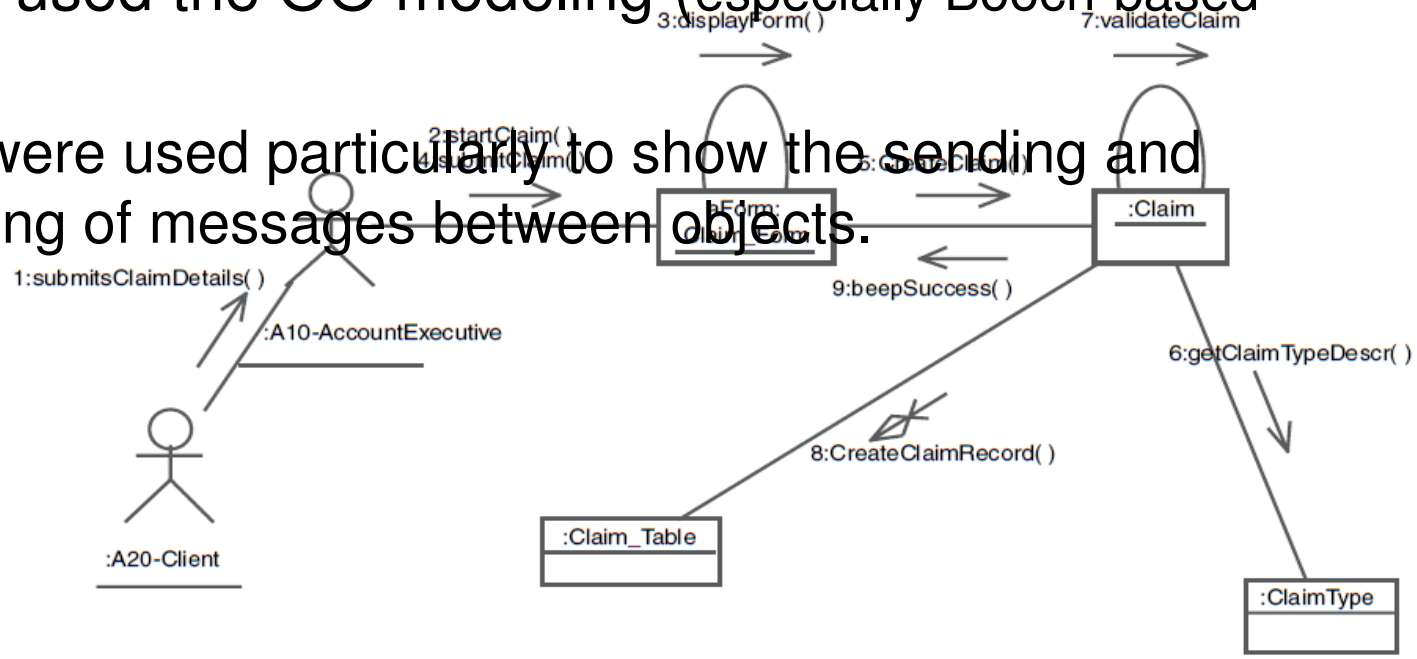
V&V of the MOSS

- Sequence diagrams (advanced)
  - ✓ Support and augment the classes created in the solution space by adding to and cross-checking operations in a class diagram that are used in implementing messages and their sequences.



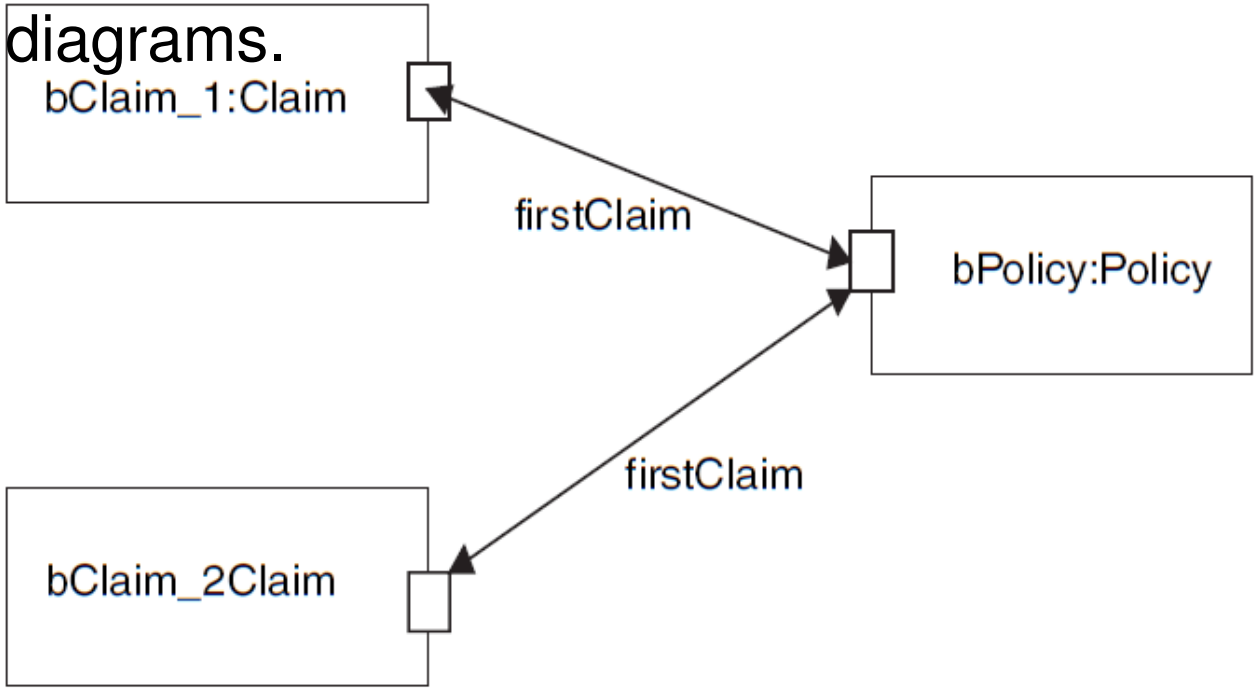
V&V of the MOSS

- Communication diagrams
  - ✓ Provide information similar to that of sequence diagrams.
  - ✓ These diagrams are favored by many system designers, who have used the OO modeling (especially Booch-based designers).
    - ✘ They were used particularly to show the sending and receiving of messages between objects.



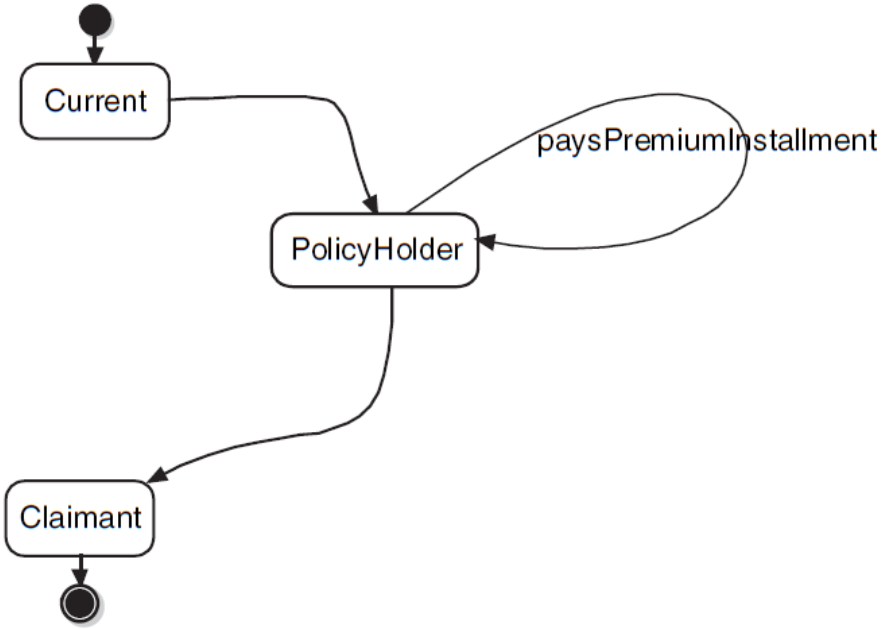
V&V of the MOSS

- Object diagrams
  - ✓ Provide a snapshot of how various objects will be linked with each other at run-time and at particular points in time.
  - ✓ Object diagrams are also helpful in analyzing multiplicities shown on class diagrams.



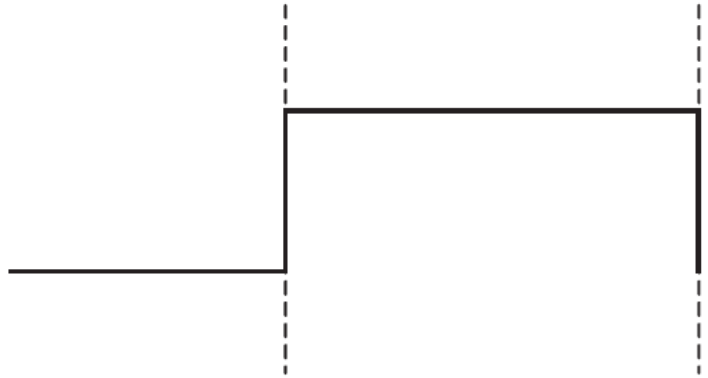
V&V of the MOSS

- State machine diagrams (advanced)
  - ✓ Show the state changes of an object, thereby depicting the object's life cycle.
  - ✓ They are extremely relevant in modeling real-time systems.



V&V of the MOSS

- Timing diagrams
  - ✓ Depict the state changes of one or more objects, thereby also facilitating comparison between state changes to different objects.



Analyzing MOPS for MOSS

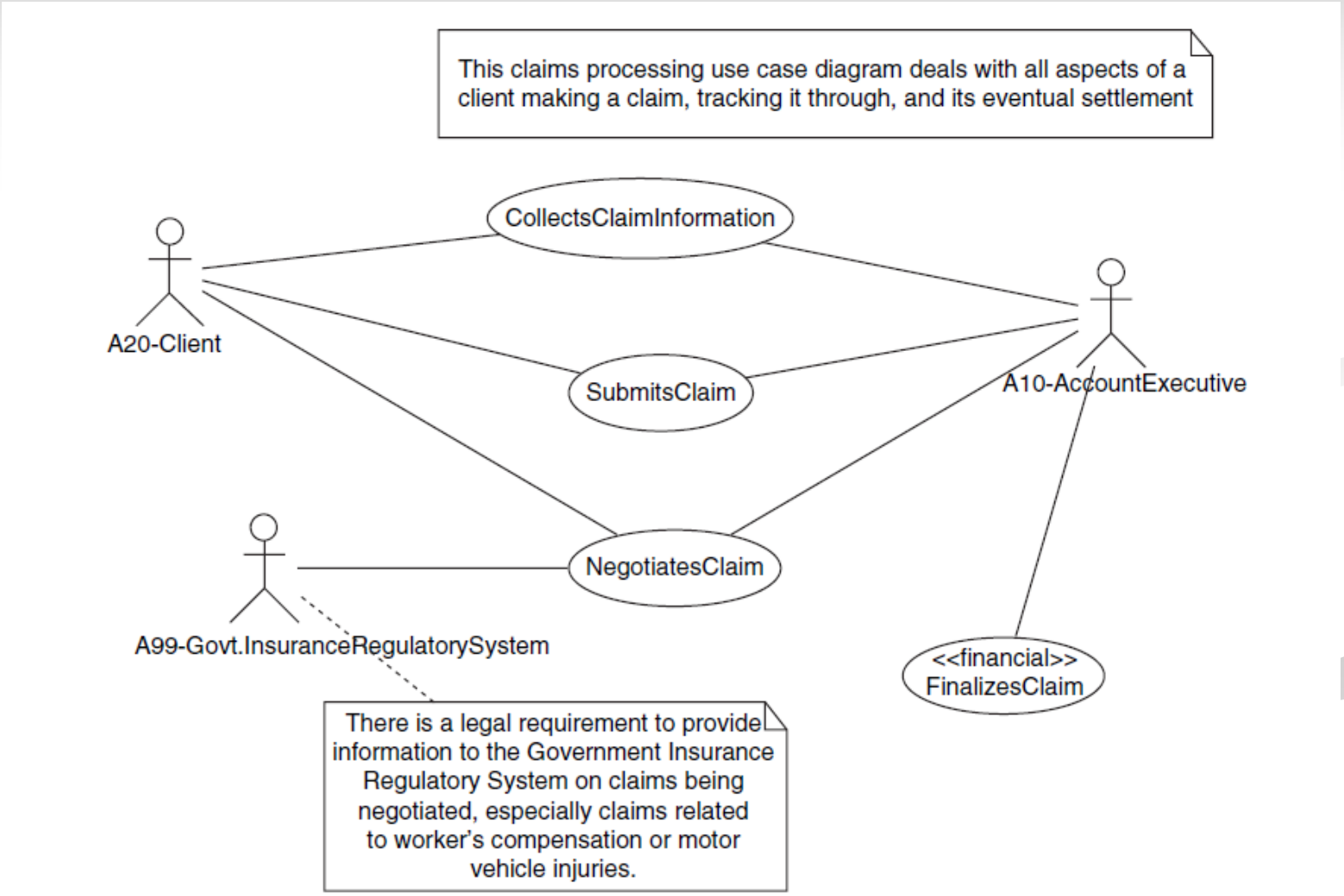
- The modeling effort in the solution space builds on the models already created in the problem space.
  - ✓ Analysis of MOPS is an ideal starting point for creating and verifying the quality of MOSS.
- Three main analysis
  - ✓ Analysis of Use Cases in the Solution Space
  - ✓ Analysis of Class Diagrams in the Solution Space
  - ✓ Analyzing Activity Diagrams in the Solution Space

Analysis of Use Cases in the Solution Space

- The use case diagrams, and their corresponding use cases, are analyzed to ensure that they are supported by the solution
  - ✓ "Are we building the product right?"
  - ✓ The process for this analysis is described as the business analysis process component.
- This analysis reveals implementation-level information for a class such as attribute types and operation signatures.
  - ✓ The focus, in stepping through the use cases in the solution space, is to ensure that the implemented solution is able to handle the functionalities specified in the use cases, an exercise embedded the V&V process.

Analysis of Use Cases in the Solution Space

- Example



Analysis of Class Diagrams in the Solution Space

- The class diagrams are extended in MOSS with the aim of converting them into implementable entities.
  - ✓ MOPS creates class diagrams to document business entities, their attributes and their behaviors.
  - ✓ The actors, use case documentation and even notes attached to the use case diagrams are analyzed to get a complete picture of the classes required and additional information on the classes already discovered to facilitate their implementation.

Analysis of Class Diagrams in the Solution Space

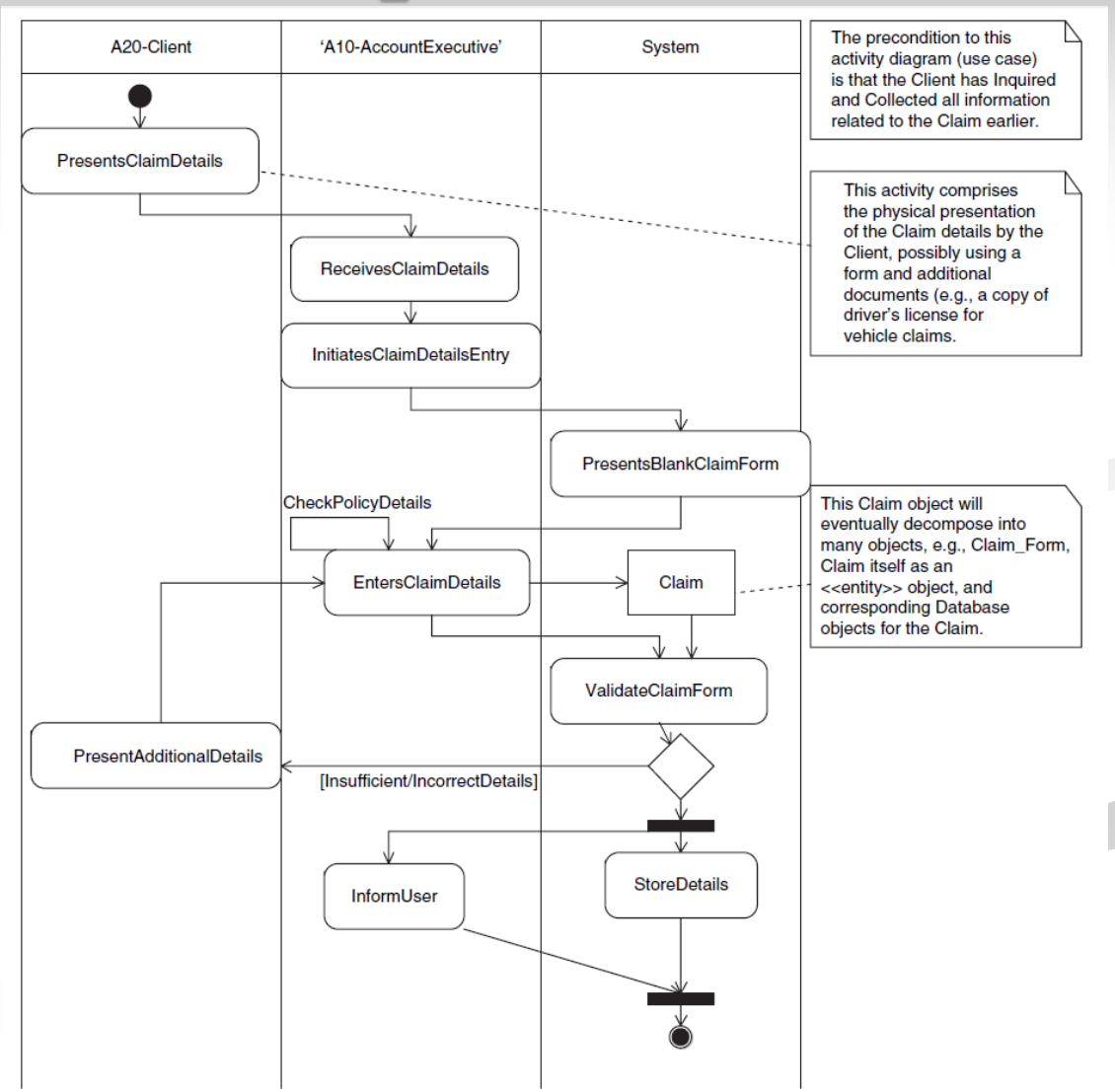
- This qualification and definition of classes and class diagrams in the solution space will result in the following:
  - ✓ Additional attributes and operations inside the classes enabling implementation and execution of the classes.
  - ✓ Additional information about the attributes and operations (such as attribute types, their initial values, and operation signatures) that are necessary for the implementation of a class.
  - ✓ Creation of GUI and database classes that will enable display and storage of information.
  - ✓ Refinement of classes by capitalizing on the concepts of quality and reuse promulgated by object orientation.

Analyzing Activity Diagrams in the Solution Space

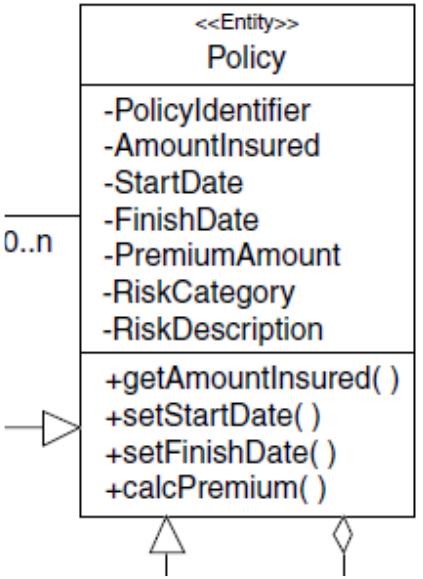
- The activity diagrams drawn in the problem space are further analyzed by system designers to ensure that the activities documented in MOPS have corresponding implementation-level constructs in the solution classes.

Analyzing Activity Diagrams in the Solution Space

- Example



Syntax Checks for Classes in the Solution Space



<<Entity>>
Policy

-<<identifier>> PolicyIdentifier: Integer = 00
-AmountInsured : Currency = 00
-StartDate : Date = 01/07/1984
-FinishDate : Date = 00/00/0000
-PremiumAmount : Currency = 5000
-<<flag>>Status : Byte = 0

+getAmountInsured (PolicyIdentifier : Policy) : Boolean
+setStartDate (PolicyIdentifier : Integer, StartDate : Date) : Boolean
+setFinishDate( ) : Boolean
+calcPremium (PolicyIdentifier : Integer, PremiumAmount : Boolean) : Boolean
+getPolicy (PolicyIdentifier : Policy, p : Policy) : Boolean
+<<database>> storePolicy (p : Policy) : Boolean
+getInterestRate( )
+checkRenewalOrNew( )

Syntax Checks for Classes in the Solution Space

- Type of syntax checks
  - ✓ Class: Names, Stereotypes, Type
  - ✓ Attributes: Types, Initial Values, Visibility, Stereotypes
  - ✓ Operations: Signatures, Visibility, Stereotypes
  - ✓ Exception Class
  - ✓ Error Handling

Semantics Checks for Classes in the Solution Space

- Type of semantic checks
  - ✓ Meaning of Class Name
  - ✓ Meaning of Attribute Name
  - ✓ Attribute Types and Initial Values
  - ✓ Meanings of the Operations
  - ✓ Pre- and Postconditions of Operations
  - ✓ Signature of the Operation
  - ✓ Stereotypes of Operations
  - ✓ Scope of Operations
  - ✓ Overloading of Operations
  - ✓ Overriding Operations
  - ✓ Overriding Variables
  - ✓ Encapsulation

Semantics Checks for Classes in the Solution Space

- Type of semantic checks
  - ✓ Meaning of Class Name
  - ✓ Meaning of Attribute Name
  - ✓ Attribute Types and Initial Values
  - ✓ Meanings of the Operations
  - ✓ Pre- and Postconditions of Operations
  - ✓ Signature of the Operation
  - ✓ Stereotypes of Operations
  - ✓ Scope of Operations
  - ✓ Overloading of Operations
  - ✓ Overriding Operations
  - ✓ Overriding Variables
  - ✓ Encapsulation

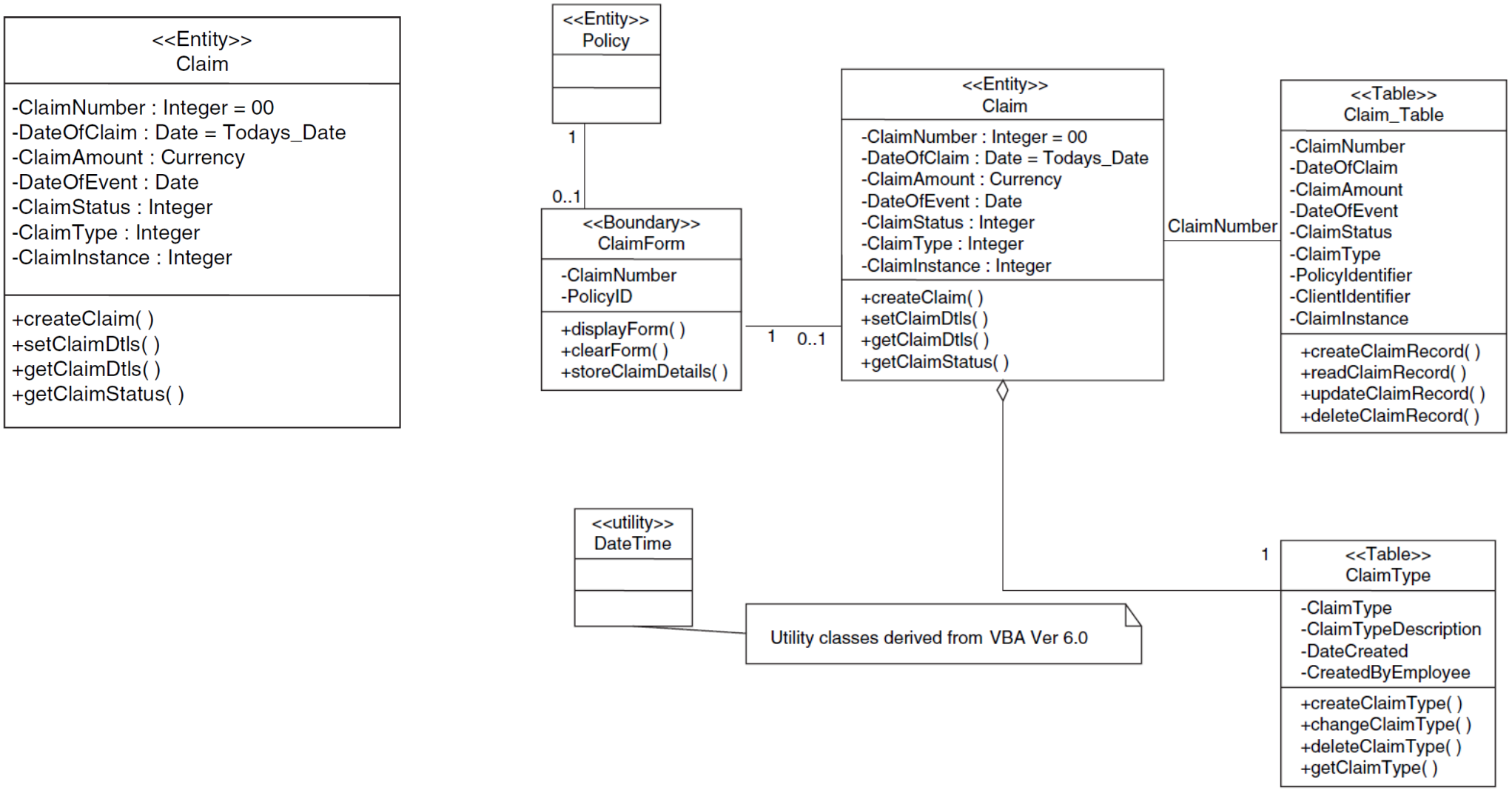
Aesthetic Checks for Classes at the Design Level

- Although in the solution space one would expect two to three times the number of attributes per class in the problem space, it is still important to maintain a manageable list of attributes within a class.
- Unless absolutely necessary due to programming language needs, classes with large numbers of attributes should be refactored in order to improve their quality and readability.
- Type of checks:
  - ✓ Number of Operations
  - ✓ Load on Operations
  - ✓ Load on the Class

Describing Class Diagrams in MOSS

- After checking the quality of a class the analyzing the quality of advanced class diagrams made up of solution-level classes.
- Some classes had been discovered in the problem space, on their own they are unimplementable.
  - ✓ Additional classes that facilitate the implementation of these in the solution space are need to be discovered.

Describing Class Diagrams in MOSS



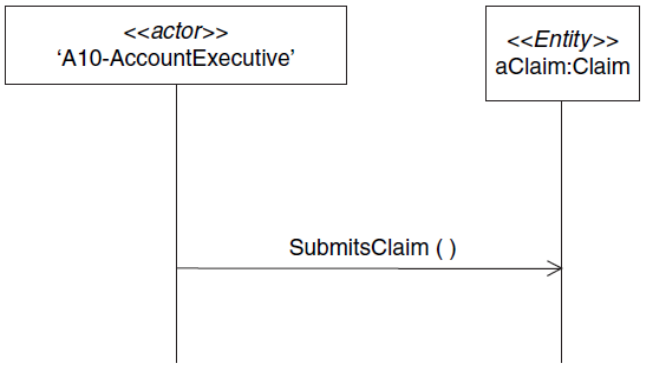
Syntax Checks of Class Diagrams in MOSS

- Class Names and Relationship Names
- Relationships
- Multiplicities
- Stereotypes
- Correct Representation of Roles on the Class Diagram
- Association of Classes with Language Libraries
- Package-Level Visibility for Classes in Packages

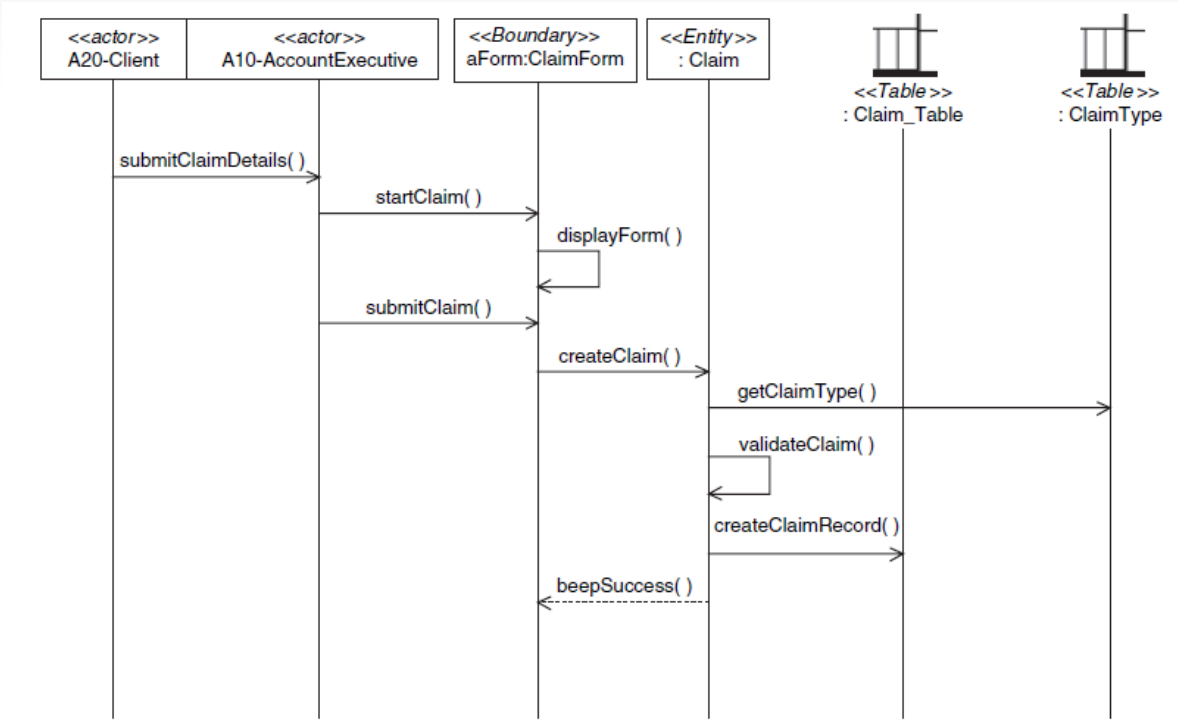
Semantic Checks of Class Diagrams in MOSS

- The Meaning of the Entire Class Diagram from the Implementation Viewpoint
- The Meanings of the Relationships on a Class Diagram
- Directions of Association
- Check for Collection Classes
- Roles for Classes
- Business Rules Behind Multiplicities
- Check for Association Classes
- Check for Overloading of a Specialized Class
- Check for Encapsulation
- Check for Reuse
- Language Constructs for Implied Meaning

Quality of sequence diagrams in MOSS



MOPS



MOSS

Syntax Checks for Sequence Diagrams

- Correctness of Objects on Sequence Diagram
- Correctness of Actors on Sequence Diagram
- Correct Sequence of Operations
- Object-to-Object Interaction
- Message Types Shown in Sequence Diagram
- Return Protocols
- Syntax of Message Signatures and Return Values
- Syntax for Multiple Messages
- Check for Multiple Objects on Sequence Diagram

Semantic Checks for Sequence Diagrams

- Purpose or Meaning Behind Sequence Diagram
- Meaning of Focus of Control
- Check for Creation and Destruction of Objects
- Use of Patterns
- Alternative Flows

Aesthetic Checks for Sequence Diagrams

- Ensure that Interactions Are Cohesive
  - ✓ If the diagram deals with more than one subject area, it is highly advisable to split it into two or more.
- Number of Objects and Messages
  - ✓ Up to 20 messages may be acceptable in a complex sequence diagram, although 12 to 15 are preferred in the solution space.
- Check for Notes and Annotations

Syntax Checks for Communication Diagrams

- Check the correctness of all objects and messages on the communication diagram.
  - ✓ Objects should be represented by object names, followed by class names
- Ensure that all messages are correctly numbered.
- If actors are shown on the communication diagram, check their correctness
- Check that the numbered message sequence is syntactically correct.
- Check object-to-object interaction, especially the creation of one object by another, as well as its destruction afterward.

Syntax Checks for Communication Diagrams

- Check the message types appearing on the communication diagram.
- Check the correctness of the return protocols and ensure that they match the operations specified in the classes.
- Check the syntax of the message signatures and return values.
- Check the syntax for multiple messages.
  - ✓ Multiple or iterative messages are shown by a * before the message.
- Check for representation of multiple objects.
  - ✓ It will be a tiled object.

Semantic Checks for Communication Diagrams

- Check the purpose or meaning behind the communication diagram.
  - ✓ If this diagram is drawn independently of the sequence diagram, its purpose is likely to be more technical.
- Ensure that the communication diagram represents a cohesive sequence that would
  - ✓ Example: Manage the storage or retrieval of data in a database.
- The semantic meaning behind each message in a communication diagram corresponds to the meaning of each operation of a class on the class diagram.
  - ✓ Check if a return protocol is implied or if it needs to be shown explicitly.

Semantic Checks for Communication Diagrams

- Check if the communication diagram depicts creation and destruction of objects.
  - ✓ Ensure that the objects are created at the right time and place.
- If the communication diagram based on a pattern check the pattern on which it is based to ensure that it is used correctly.
- Communication diagrams are unable to show if-then-else scenarios properly.
  - ✓ Check if there are alternative flows in a sequence and create separate communication diagrams for them.

Aesthetic Checks for Communication Diagrams

- If an object is overloaded, its work can be distributed among other objects or a new object (and its corresponding class) can be introduced.
- Ensure that the communication diagram shows a cohesive set of interactions between collaborating objects.
  - ✓ If the diagram attempts to deal with more than one subject area, it is highly advisable to split it into two or more.
- Ensure that the communication diagram has sufficient notes and other annotations to explain its technical aspects.
- Ensure that the communication diagram has neither too many nor too few objects (similar to sequence diagrams).

Syntax Checks for Object Diagrams

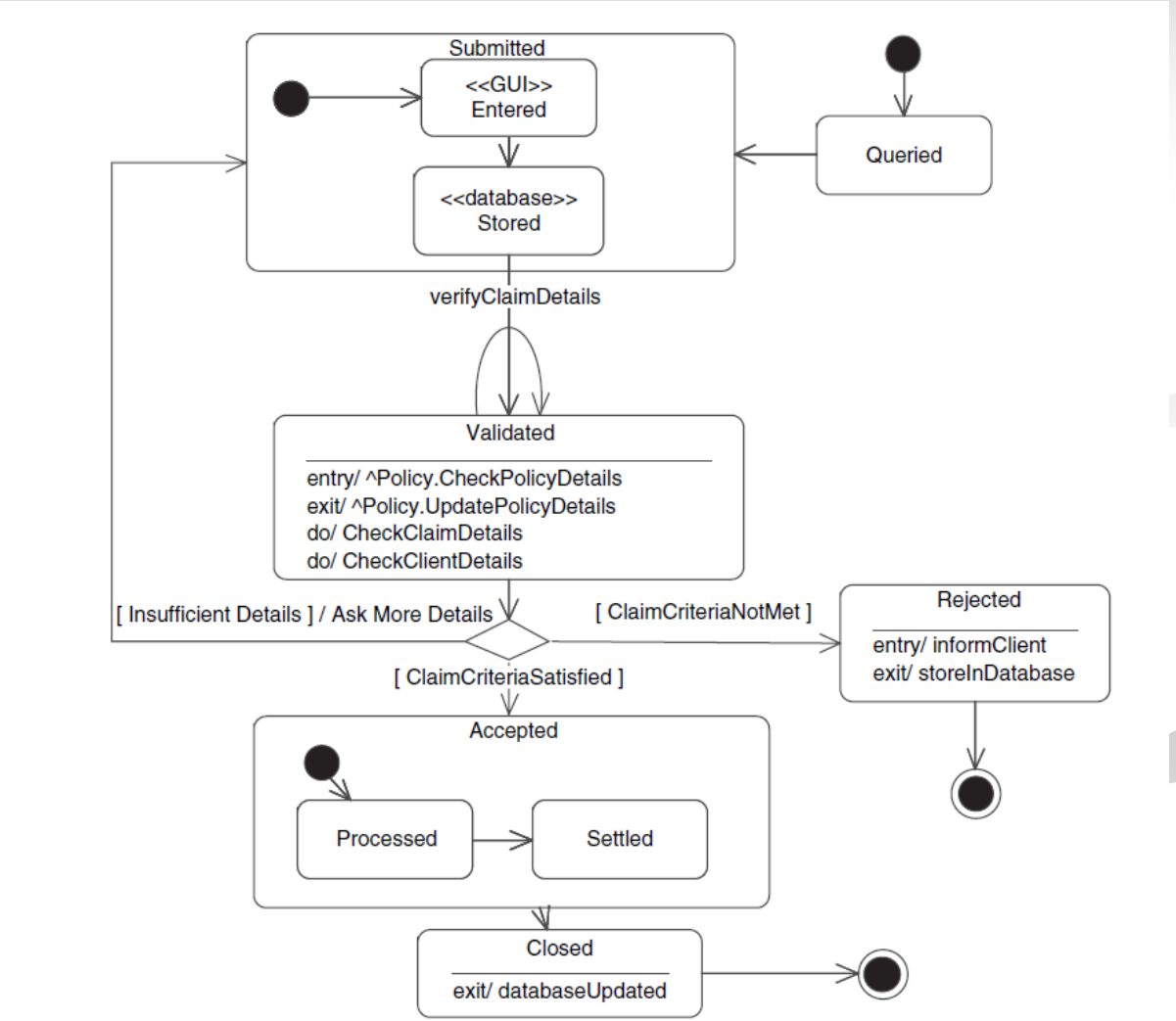
- Check the correctness of representation of objects.
- Object relationships are shown by links, which are straight lines between objects.
  - ✓ Links can be stereotyped, but more often than not, a link indicating a direct relationship between two objects is sufficient to express the relationship.
- Check that only one object is represented in a rectangle.
- Attributes or operations cannot be shown in object diagrams.
- Object diagrams do not have multiplicities shown as numbers.
  - ✓ This is because multiplicities of class diagrams are converted into actual numbers of objects shown on the object diagram itself.
- Check if notes were provided for additional explanation of the diagram.

Quality of state machine diagram in MOSS

- While state machine diagrams drawn in the solution space are detailed and technical, they are helpful for classes that come from the problem space.
- State machine diagrams for pure implementation classes (like lists and arrays) may not be very helpful even in the solution space.
- However, states for database tables and rows can add quality to models in the solution space.

Quality of state machine diagram in MOSS

- Example



Syntax Checks for State Machine Diagrams in MOSS

- Check for correct representation (notation) of states on the diagram.
- Check for the correctness of transitions between states.
- Check that the events that start the transitions are correctly specified on the diagram.
- Check that the guard conditions that start the transitions are correctly represented within [ ] brackets.
- Check if there are entry conditions to the state and, if so, that they are correctly represented.

Syntax Checks for State Machine Diagrams in MOSS

- Check for exit conditions out of the state.
- Are there ongoing actions to be performed while the object is in a particular state?
- If so, it should be represented by a ‘Do/’ within the specifications of the state.
- Are there activity states on the diagram and, if so, are they correctly represented?
- Check for existence of start and stop states on the diagram.
  - ✓ Are there action states on the diagram and, if so, are they correctly represented?

Semantic Checks for State Machine Diagrams in MOSS

- Check the meaning behind the state machine diagram.
  - ✓ Is it simply the state of an object, or is it the state of the system that is being shown?
- Check if there are messages going to other objects in the system.
  - ✓ This should be shown only if absolutely necessary. Otherwise, notes should be used to represent such messages.
- Check if messages are being received from other objects.
- Check if the diagram needs to show nested states.
  - ✓ In the solution space, it may be necessary to show nesting between complex states.

Semantic Checks for State Machine Diagrams in MOSS

- Check if historical states are shown on the diagram and whether they are meaningful.
- Check the meanings behind parallel states on the diagram, if shown.
- State machine diagrams should map with objects. These objects may be representative of a class within a class diagram.
  - ✓ Check the mapping between objects on a state machine diagram and with classes on a class diagram.

Aesthetic Checks for State Machine Diagrams in MOSS

- The number of states on a diagram, and their complexity, should be understandable.
  - State machines with more than 10 states may not be elegant, and more than one state machine diagram may have to be created.
- For large, complex and nested states, additional state machine diagrams should be considered.
- Adding notes to explain the states and transitions will enhance the readability of these diagrams.

Converting Model into Systems

- One important aspect of quality in UML-based projects is the conversion of the model into its corresponding software.
  - ✓ This is a crucial aspect of quality assurance, because a good conversion or a good implementation depends on a good model and a good process.
- In any implementation it is also important to consider how the models are converted into code.
  - ✓ Quality control or testing would appear in the final stages of a software life cycle.
  - ✓ Good quality assurance will ensure that this conversion is process based and has been prototyped properly.

Converting Model into Systems

- Models are by necessity incomplete.
  - ✓ If they comprised the actual implementation, there would be no need for modeling.
  - ✓ Because of this necessary incompleteness, a certain amount of skill is needed in converting the models into their implementation.
- Good quality assurance will ensure that this conversion is process based and has been prototyped properly.
- There are checks and balances at the end of each step within the conversion of the models into their corresponding implementation before a full release is produced.

Cross-diagram Dependencies

- Class and Sequence Diagrams
  - ✓ Operations in the class diagram have a direct mapping with the messages shown on the sequence diagram.
  - ✓ Semantic and aesthetic checks ensure that the dependencies between the two diagrams have been checked for message-operation correlation.
  - ✓ Objects of sequence diagrams should belong to classes on class diagrams.
    - × An object without a corresponding class should be investigated for its meaning and for its existence, for that matter.

Cross-diagram Dependencies

- Activity and Sequence Diagrams
  - Typically, a thread running through one suite of activities within an activity diagram can be represented by one sequence diagram.
    - It is advantageous to check for relationships between activity diagrams and sequence diagrams.
  - The semantic meaning behind a thread through an activity diagram and the corresponding sequence diagram should be established.
  - One activity diagram may relate to many sequence diagrams.
    - This should be checked to ensure that aesthetically they are all properly related and that it is possible to track these diagrams easily.

Cross-diagram Dependencies

- Class to State Machine Diagrams
  - A class on a class diagram is related to the life cycle of its corresponding object on a state machine.
  - It should be possible to establish the link between the class and a corresponding state machine.
- Sequence to Communication Diagrams.
  - These two diagrams are merely views of each other.
    - For an aesthetically elegant solution, both diagrams may not be required.
  - The need for the diagrams will depend on the interests and needs of the modelers.

Cross-diagram Dependencies

- State Machine to Timing Diagrams
  - Timing diagrams have the potential not only to show the states, but also to enable their comparison between multiple objects.
    - These objects may have corresponding state machine diagrams that show the state changes and transitions in great detail.
  - Together, the two diagrams provide a complete and comparative picture of objects, states and transitions.
  - The semantic meanings behind the objects on the two diagrams should be very easy to establish - merely by checking the names of the objects.
    - The aesthetic check can include checking for the number of objects shown on the timing diagram and any additional notes to clarify the cross-diagram dependency.

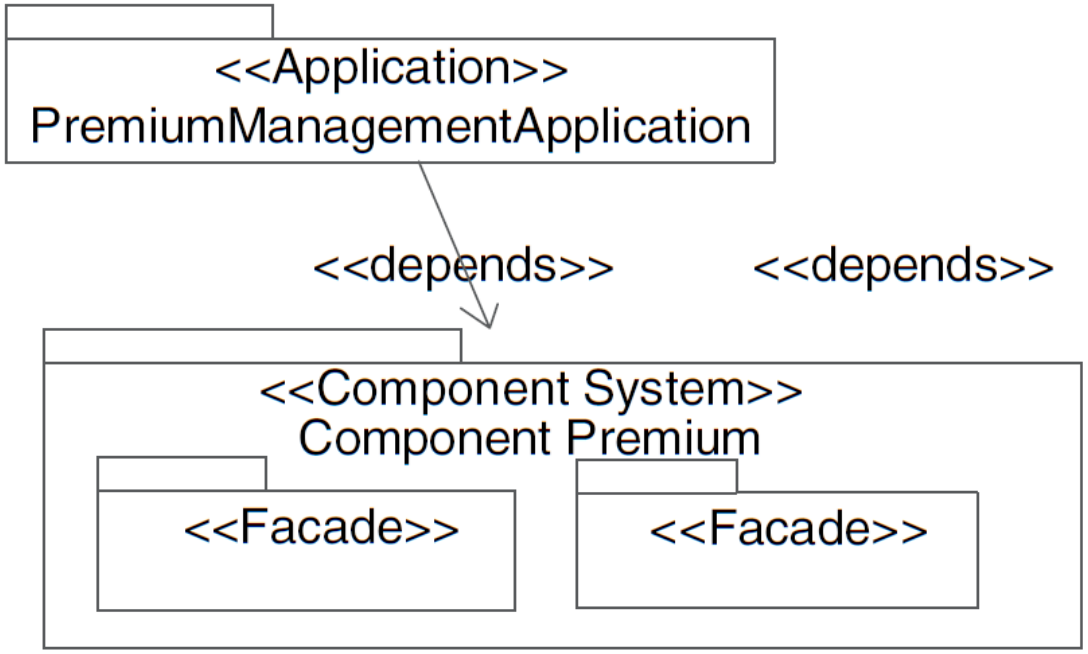
MOBS

V&V of the MOBS

- The UML diagrams in background modeling space include:
  - ✓ package diagrams that help organize the information architecture;
  - ✓ class diagrams that deal with persistence (storage and retrieval of data);
  - ✓ robustness diagrams that introduce control classes to ensure robustness of architecture;
  - ✓ component diagrams including architectural diagrams for Web design or Web application servers;
  - ✓ composite structure diagrams that show run-time decomposition of components or objects;
  - ✓ the deployment diagram, which represents the deployment of hardware and its mapping with the components.

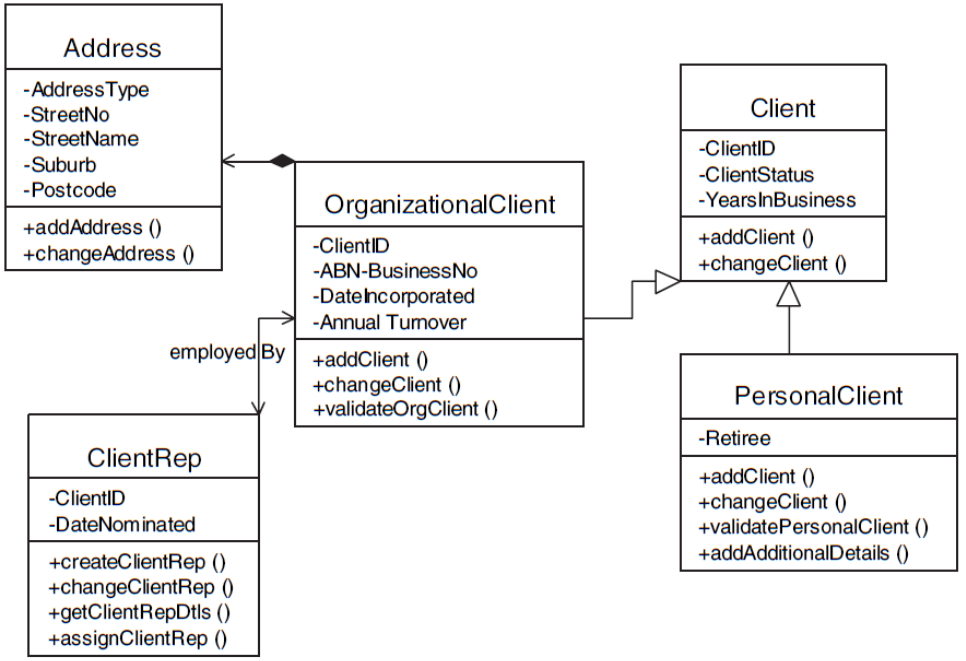
V&V of the MOBS

- Package diagrams
  - ✓ They are an extension to the package diagram drawn in MOPS.
  - ✓ In this architectural space, they include packages for implementation including packages representing databases and GUIs.



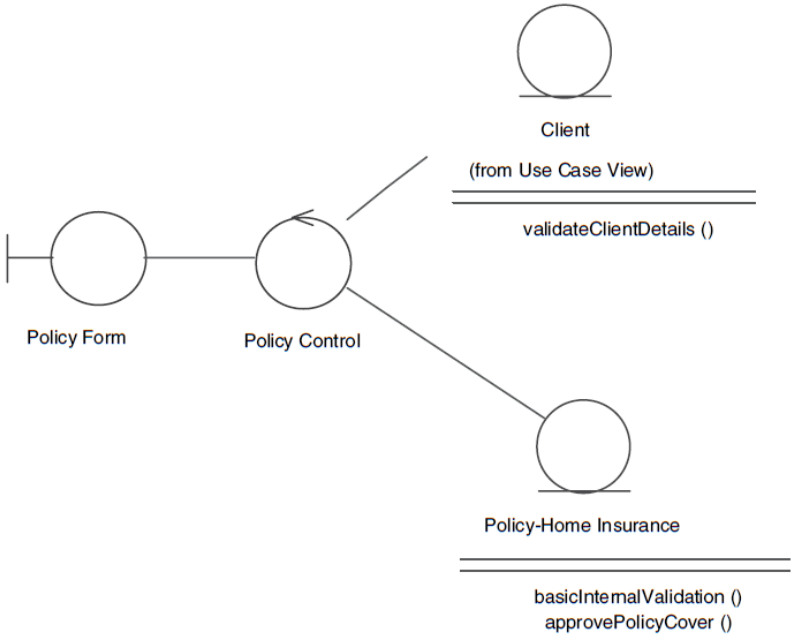
V&V of the MOBS

- Class Persistence diagrams
  - ✓ They extend and enrich the class diagrams drawn in the other two modeling spaces to ensure database modeling and implementation.
  - ✓ They consider database mapping, active classes, granularity and reuse and related issues.



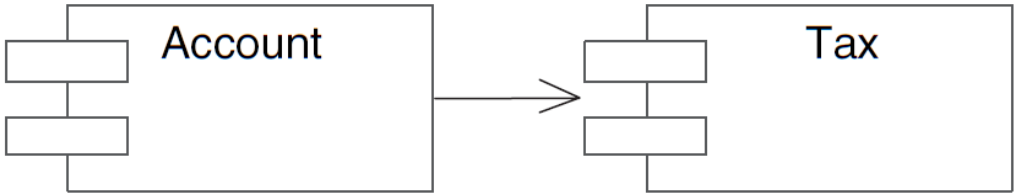
V&V of the MOBS

- Robustness diagrams
  - ✓ They add value to the class diagrams drawn in the background space by separating the entity classes from the boundary classes and from one another, ensuring robustness and flexibility of designs.
  - ✓ The robustness is ensured by inserting control classes at the right place within the class diagrams.



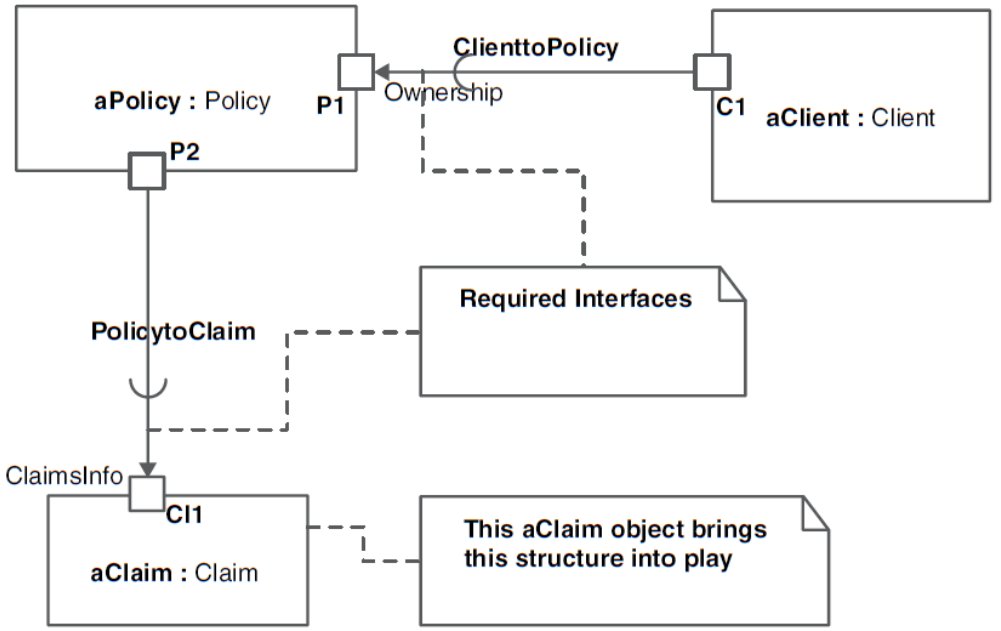
V&V of the MOBS

- Component diagrams
  - ✓ They are drawn in the background space as a major architectural constituent of the system.
  - ✓ These are the physical implementation diagrams in UML, depicting where and how the classes designed thus far are to be implemented.



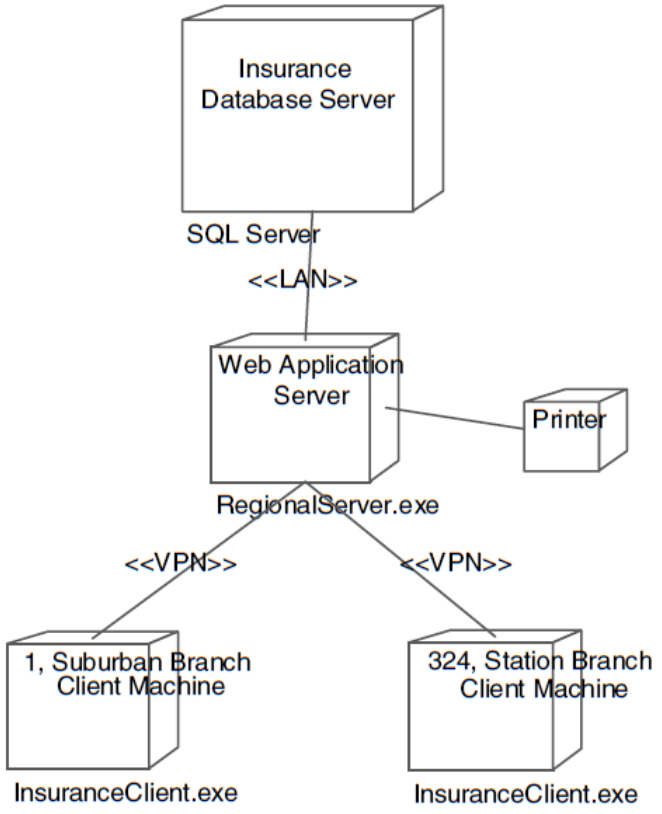
V&V of the MOBS

- Composite structure diagrams
  - ✓ They model the run-time decomposition of a component or an object.



V&V of the MOBS

- Deployment diagrams
  - ✓ The only hardware specific diagrams in UML.
  - ✓ The are drawn in the background space to outline the physical deployment of the system, including its nodes and processors.



Functional vs. Infrastructure

- The brief initial requirements in MOPS provide information that helps identify and evaluate packages.
  - ✓ It is these packages that are refined and embellished in this background space by the creation of layers and corresponding architectural prototypes resulting in overall fitness of a package in the system.
- The application layers emerge from the functional descriptions of the system attempted in the problem space.
  - ✓ Layers are conceptual in nature and deal with the architecture of the system.
  - ✓ Layers are quite different from the classes with the same names.

Functional vs. Infrastructure

- it is discovered that the business processes do not remain contained within one package.
  - ✓ As a result of the influence of the problem space on the background space.
  - ✓ Businesses modeled on processes (rather than departments) are unlikely to be compartmentalized in packages.
    - × Some business processes transcend many packages.
- Keeping this in mind is helpful as we create packages, because although packages represent subsystems, these subsystems will have dependencies on each other.
  - ✓ The business process aspect of good-quality architecture must be kept in mind in arriving at the application- versus infrastructure-based divisions of the system

Functional vs. Infrastructure

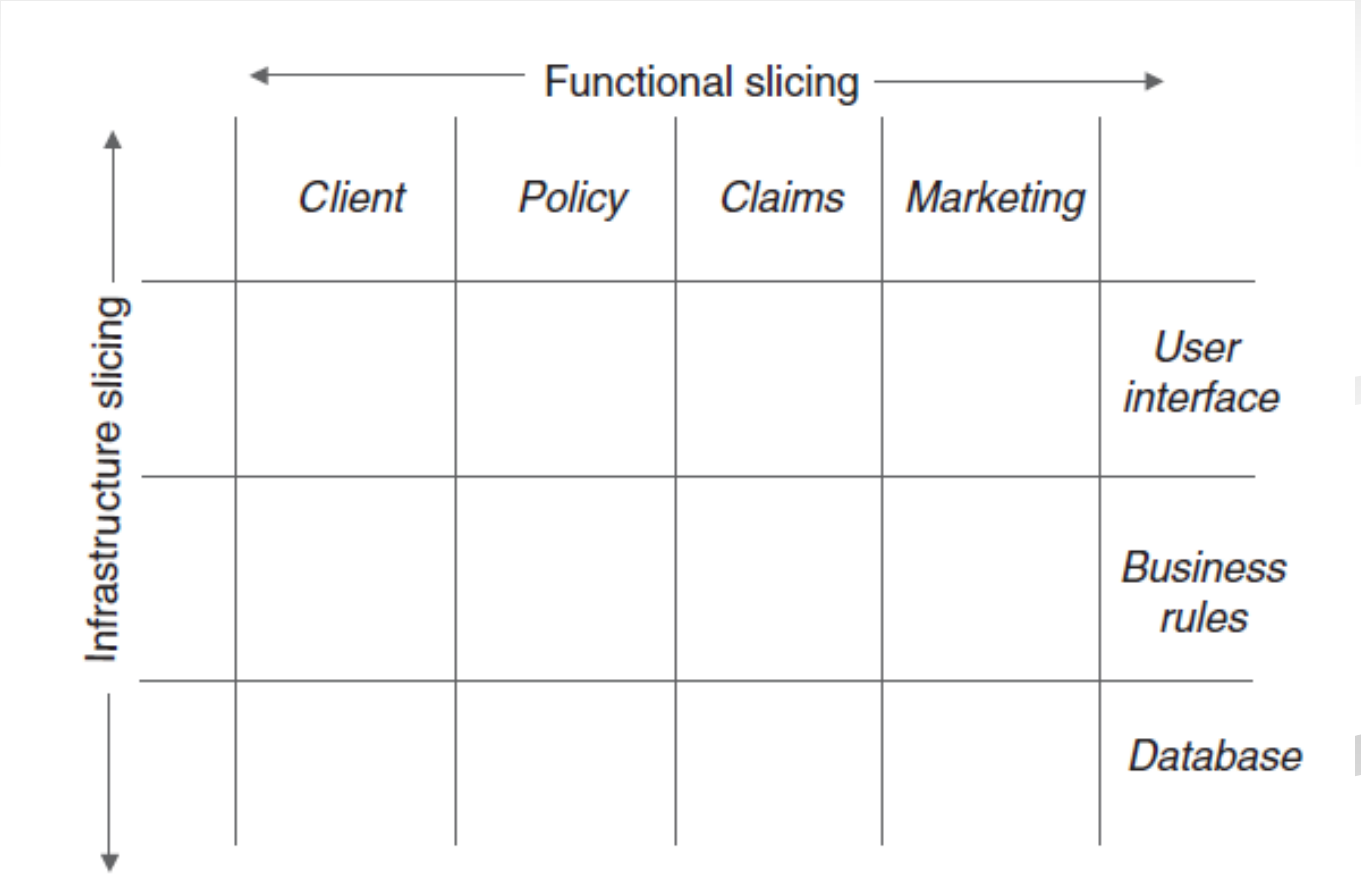
- Modeling in the background space may reveal major which will influence the models representing the problem domain (the influence of MOPS on the background architecture).
  - ✓ Examples: limited bandwidth or unavailability of a content management system.
- It is essential to consider influence from MOSS on the background architecture, or solution space models, on the architecture of the system.
  - ✓ From a solution/implementation perspective, each of the business packages will need at least three slices of technical infrastructure to carry out its responsibilities.

Functional vs. Infrastructure

- Getting the overall architecture of the system right is one of the most significant steps toward achieving quality.
- The architectural work also influences iterative and incremental project planning.
  - The development project plan can contain iterations that facilitate creation of software artifacts inside these packages, whereas the incremental aspect of the project plan would include development of newer packages.

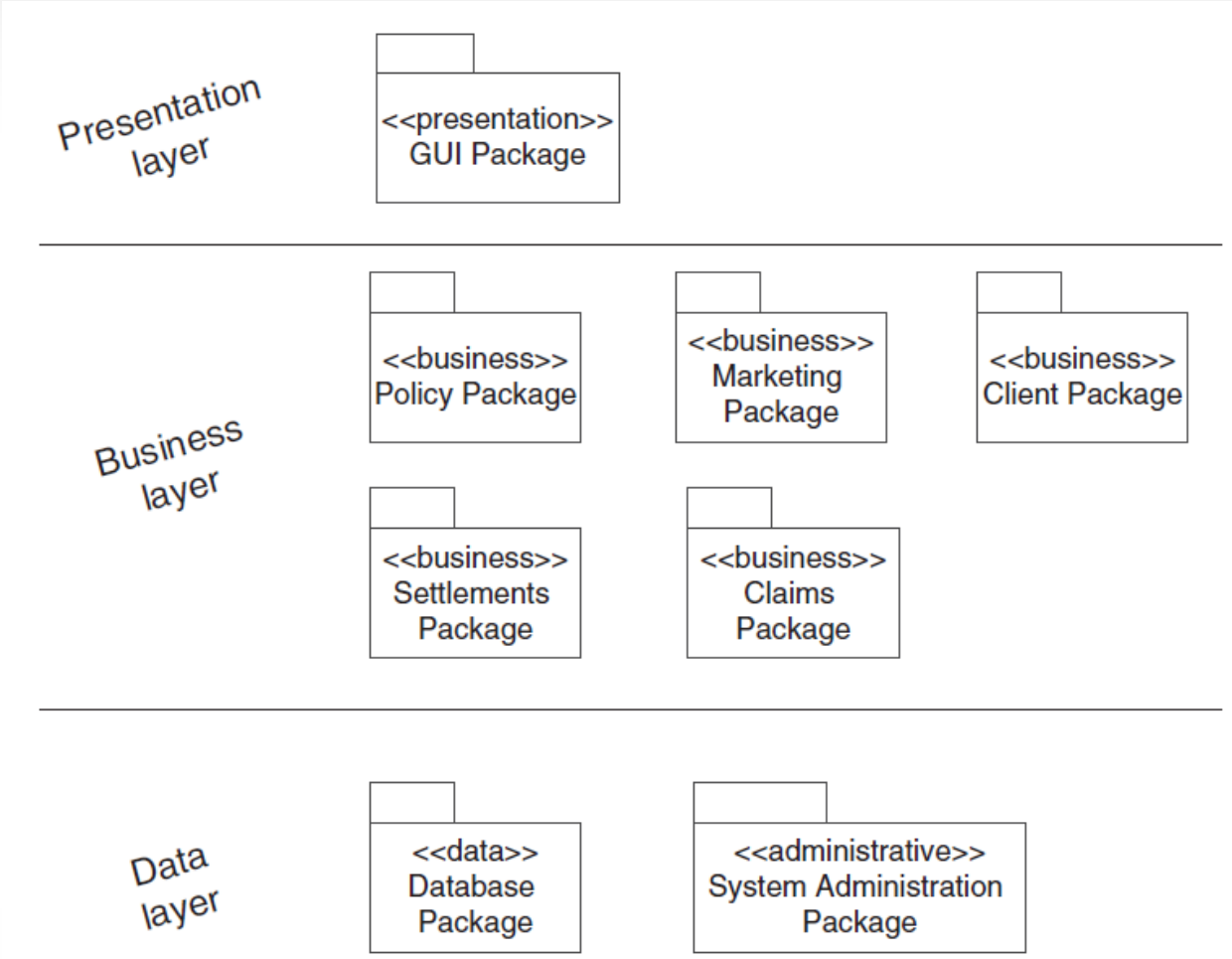
Functional vs. Infrastructure

- Example



Functional vs. Infrastructure

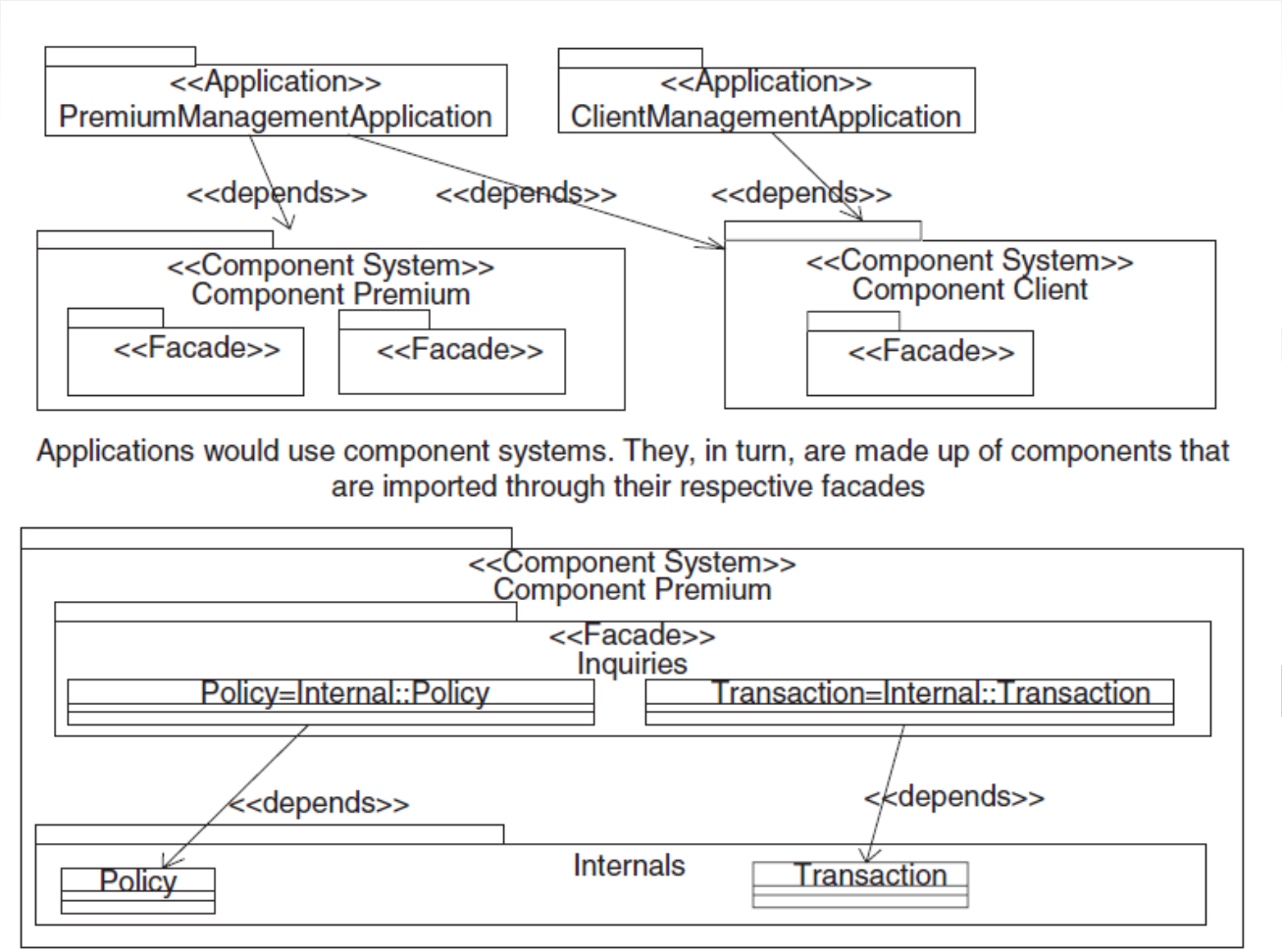
- Example:



MOPS

Functional vs. Infrastructure

- Example:



MOBS

Syntax Checks for Package Diagrams in MOBS

- Packages created as organizational units should be represented with a tabbed rectangle.
- The notation for a package, the stereotype of a package and, optionally, the relationship between packages should be checked for syntactic correctness.
- The relationship between packages, if shown, should be the dependency relationship (also optionally with a stereotype).
- Stereotypes on packages should use the correct stereotype notations.

Semantic Checks for Package Diagrams in MOBS

- Investigate the use cases and activity diagrams within each of the packages.
  - ✓ Verifying the use cases within the context of the packages is part of the background space semantic checks for package diagrams.
  - ✓ Semantic checks of packages can thus be helpful in refactoring of packages into additional smaller yet cohesive packages.
- Check the meanings behind nonfunctional packages like *GUI*, *Database* and *Administration*.
  - ✓ The focus should be on the documentation of these packages, as well as on the class and sequence diagrams underneath these technological packages.
- Check for dependencies between the packages
  - ✓ These dependencies can provide valuable input to the project manager in terms of scheduling the iterative and incremental development project plan.

Semantic Checks for Package Diagrams in MOBS

- Ensure that if packages are shown, they correspond to layers (or stereotyped as such) and they are the correct layers.
  - ✓ The GUI package is correctly stereotyped as «presentation» - alternatively, «boundary» or « interface» would also do.
  - ✓ The entity packages are stereotyped as «business», although «entity» would also be acceptable.
- Check for the extent to which the package diagrams represent the slices or tiles discussed earlier.
  - ✓ Do package diagrams include system integration (i.e., one system talking to another) models ?
  - ✓ If so, are those integration aspects mentioned on the package diagrams (through a note)?

Semantic Checks for Package Diagrams in MOBS

- Should the *XXX* application be a separate application?
- Does the *YYY* application genuinely depend on the component *AAA* and the component *BBB* ?
- What sort of interdependency there is between the component *CCC* and the component *DDD* ?
- What does the *ABC* package do?
- Is *XYZ* a genuine framework or is it a high-level pattern that will need a fair amount of implementation-level detail?
- Is the stereotyping of each of these packages correct?
- Are the components reusable?
- Are the application packages worth stereotyping as applications?
- Is *BCD* the only package inside *EFD* that deals with the actual transaction of inquiry or are other packages needed? Are other classes needed as well?

Syntax Checks for Class Diagrams in MOBS

- Check the representation of implementation constructs (e.g., mapping database storage tables to classes) in the class diagram here.
- Check the access keys and IDs represented in the classes that represent the tables. These would be the primary and foreign keys, and their correctness needs to be verified.
- Consider the representation of access keys and IDs from the point of view of language of implementation.
- Check for the way data are searched and sorted from an architectural viewpoint, and check how they are stored and accessed through the keys, indexes and definitions of classes in MOBS.

Syntax Checks for Class Diagrams in MOBS

- Check the storage by keys and the values of the position of the record in the database.
- Check the syntactic correctness of inheritance when classes are reused from the programming language or third-party libraries.
- Check additional language representations.
- Check multiplicities between the relationships of the classes.
- Check link or association classes for their relationship with the association line
  - ✓ Association classes are not directly connected to other classes but rather to their association line.

Semantic Checks for Class Diagrams in MOBS

- Check that what is being stored in the databases is correctly represented in the class diagrams.
- Check for unique responsibilities for classes.
  - ✓ Some classes should not be stored in the same table.
- Cross-check persistence representation with multiplicities shown on the class diagram.
- Revisit the keys and IDs, checking them for balance in terms of speed versus volume.
- Cross-check with direction of associations.
  - ✓ This will determine the direction of access.
- Check for possibilities and correctness of semantic reuse, which will imply reuse of business classes/objects/packages.

Syntax Checks for Robustness Diagrams in MOBS

- Syntax checks for the robustness diagram will follow the syntax checks for a class diagram in the background space.
  - ✓ However, the focus here is on the controller class, whose stereotype «control» should appear in the diagram.
- The icon for the controller stereotype should be visible on the diagram.
  - ✓ It is necessary to check the syntax of the controller class, its stereotype and its multiplicity, as well as its list of operations.
- In the robustness diagram, displaying of all correct stereotypes - «boundary», «control» and «entity» - is mandatory.

Syntax Checks for Component Diagrams in MOBS

- Syntax checks for component diagrams focus on correctness of physical implementation of the system.
  - ✓ The first things to check is the correct representation of a component on the diagram - this involves using correct notation for the component.
- The next check therefore would be the representation of the interfaces on the diagram.
  - ✓ Components realize interfaces.
  - ✓ Check the names of the components as well as the names of their interfaces.
- If you there are components within components, then it is necessary to show the external package or another component to which an embedded component might belong.

Syntax Checks for Component Diagrams in MOBS

- Occasionally components may have additional values or constraints shown on the component diagrams and may have their responsibilities documented.
  - ✓ In that case, check for the correctness of this additional documentation.
- In checking the name of the component, check if there is a suffix to the name that represents either a language class, a database table or a linked library.
  - ✓ If the suffix is nonstandard or relatively unknown, make sure that it is syntactically correct or that it represents a physical entity like a document, program or file, because there is only one notation representing various types of implementations.

Syntax Checks for Component Diagrams in MOBS

- Components are usually be stereotyped.
  - ✓ Check the syntax of the stereotypes.
- Components can realize the interfaces.
  - ✓ Check the relationship arrow, which is primarily a dotted line with an open arrowhead that indicates dependency.
- Components can also depend on the interfaces of other components.
  - ✓ Check the relationship arrow, which is primarily a dotted line with an open arrowhead that indicates dependency.
- Dependency can be stereotyped again in various ways.

Syntax Checks for Component Diagrams in MOBS

- An important syntactic check is to ensure that the component, which is mapped to many classes, is physically implementable.
  - ✓ More often than not, this requires not only programming knowledge and effort, which is language specific, but also creating the actual executable and running it in the given environment.
- Linking, building and executing a particular component are crucial syntax checks that ensure that the component is syntactically correct.

Semantic Checks for Component Diagrams in MOBS

- Checking the semantics of the components in MOBS is far more involved and requires a good understanding of the technological or solution domain.
- Knowledge of languages and databases is important in deciding whether or not a component has been correctly modeled semantically.

Semantic Checks for Component Diagrams in MOBS

- Check that the component correctly represents the class or collection of classes that is meant to represent.
  - ✓ It is important to create a component showing the relationship between the component and the classes (through its specifications) and to make sure that the component provides the right number of well-defined interfaces that it realizes.
- It is important to check whether a component has sufficient interfaces, the extent to which these interfaces overlap each other and how the component realizes these interfaces.
  - ✓ Ideally, interfaces should not overlap.
  - ✓ One quality of a good component is that it is easily replaceable.
  - ✓ If a component has more than four or five interfaces and if they are used right through the system, then replacing one component by another will be far more involved.

Semantic Checks for Component Diagrams in MOBS

- Use of third-party components is quite common and is encouraged.
  - ✓ The semantic meaning of each third-party component used in the project should be clarified.
  - ✓ Check the meanings of any external components used on your diagram.
    - × This involves checking the interfaces and their definitions, and the specifications of the components that realize these definitions, as provided by the vendor of the components.
- Third-party components need not be only database-related technical components.
  - ✓ They can also be business components, as would be tendered by, say, the MDA architecture.
  - ✓ Understanding the business functionality of the components is extremely important in creating a semantically complete component diagram.

Semantic Checks for Component Diagrams in MOBS

- Another major semantic check involves the number of classes belonging to or implemented by a component.
  - ✓ Mapping a class or a suite of classes to a component is an important semantic activity that needs focused verification.
- Check that a class created in either MOPS or MOSS is not floating around in those models without being realized in the background space.
- Increasing access to a legacy application in order to retrieve data is done through a legacy “wrapper” which is an interface of a component, and the entire backend code is treated as a component itself.
  - ✓ This should be checked to ensure that the legacy application is able to realize the interfaces that it provides.

Semantic Checks for Component Diagrams in MOBS

- Check to see if the technical architectural issues are satisfied by a prototyped component.
- If a prototype is created, then it is important to trace the results of the prototype to the operational requirements.
- Creation of components goes hand in hand with their ability to be reused.
  - ✓ It is important to check that components have sufficient and crisply defined interfaces that enable them to be reused in the next project and beyond.

Semantic Checks for Component Diagrams in MOBS

- The need to check all possible scenarios where the component or parts of it to be produced are likely to be reused, and to provide corresponding interfaces for these reusable components, are crucial quality criteria.
- Security architecture and security components also provide an interesting syntax and semantic challenge to quality.
  - ✓ Components that relate to security should be checked for syntactic correctness, which involves their ability to integrate with the application that is being produced and to function correctly.
  - ✓ The functional bit of the security component will be the part of semantic checks.

Syntax Checks for Composite Structure Diagrams

- Check for representation of object or components at run-time.
- Check for correct representation of the port symbol.
- Check for the *Interface* representation - a required and realized interface.
- Check for accuracy of the direction arrow.
- Check for correctness of notes notation.

Semantic Checks for Composite Structure Diagrams

- The semantic check for the composite structure diagram deals with checking whether it faithfully represents a run-time scenario.
  - ✓ It may be necessary to trace the meaning of this diagram back to other diagrams in the other two modeling spaces.
- Individual components and their meanings should also be checked for correct representation;
  - ✓ However, that can be expected to follow the semantic checks for object diagrams and communication diagrams.

Cross-diagram Dependencies in MOBS

- Package and Component Diagrams
  - When components are shown within packages, there is a dependency between the two diagrams.
    - Semantic checks relating to the meaning of the packages and the components within them, as well as aesthetic checks focusing on the number of components within a package, should be performed here.
  - Although technically there is no limitation on the number of components on a package, the size of the package must be kept in mind.
    - If a package is made up of a large number of components (e.g., 20 components in a typical Java or C++ implementation), then further layering of packages must be considered.

Cross-diagram Dependencies in MOBS

- Class and Component Diagrams
  - ✓ Components realize classes. There is a certain mapping between classes and components.
    - × This is not shown visually in most CASE tools.
  - ✓ Components to class mapping must be investigated at both semantic and aesthetic levels in V&V in MOBS.
    - × A component can realize 7 to 10 «entity» classes.
  - ✓ Stereotypes of classes can appear in a component.
  - ✓ Separate components realizing the user interfaces and the databases must be considered to enhance the quality of the architecture.

Cross-diagram Dependencies in MOBS

- Composite Structure and Object/Communication Diagrams
  - ✓ The run-time components shown on communication diagrams are also run-time objects for all practical purposes (although in theory objects and run-time components are different).
  - ✓ In composite structures, dependencies between run-time components are shown, as against the messages in component diagrams.
  - ✓ Composite structure diagrams show interfaces, which are not shown on communication diagrams.
  - ✓ A cross-diagram check between communication/object diagrams and the corresponding composite structure diagrams should be considered valuable in improving the semantic and aesthetic quality of MOBS.

Cross-diagram Dependencies in MOBS

- Component and Deployment Diagrams
  - Components eventually execute on processors represented in the deployment diagrams.
  - It is important to investigate, again at the semantic and aesthetic levels, the number of components executing on a processor.
    - This can lead to interesting discussions about the load and distribution of components at run-time.
  - It also leads to enhancement of security and connectivity in the architecture

Questions?