

Olá,

Criei estas apostilas a mais de 5 anos e atualizei uma série delas com alguns dados adicionais. Muitas partes desta apostila está desatualizada, mas servirá para quem quer tirar uma dúvida ou aprender sobre .Net e as outras tecnologias.

Perfil Microsoft: <https://www.mcpvirtualbusinesscard.com/VBCServer/msincic/profile>

Marcelo Sincic trabalha com informática desde 1988. Durante anos trabalhou com desenvolvimento (iniciando com Dbase III e Clipper S'87) e com redes (Novell 2.0 e Lantastic).

Hoje atua como consultor e instrutor para diversos parceiros e clientes Microsoft.

Recebeu em abril de 2009 o prêmio **Latin American MCT Awards** no MCT Summit 2009, um prêmio entregue a apenas 5 instrutores de toda a América Latina (<http://www.marcelosincic.eti.br/Blog/post/Microsoft-MCT-Awards-America-Latina.aspx>).

Recebeu em setembro de 2009 o prêmio **IT HERO** da equipe Microsoft Technet Brasil em reconhecimento a projeto desenvolvido (<http://www.marcelosincic.eti.br/Blog/post/IT-Hero-Microsoft-TechNet.aspx>). Em Novembro de 2009 recebeu novamente um premio do programa IT Hero agora na categoria de especialistas (<http://www.marcelosincic.eti.br/Blog/post/TechNet-IT-Hero-Especialista-Selecionado-o-nosso-projeto-de-OCS-2007.aspx>).

Acumula por 5 vezes certificações com o título **Charter Member**, indicando estar entre os primeiros do mundo a se certificarem profissionalmente em Windows 2008 e Windows 7.

Possui diversas certificações oficiais de TI:

- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2008
- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2005
- MCITP - Microsoft Certified IT Professional Windows Server 2008 Admin
- MCITP - Microsoft Certified IT Professional Enterprise Administrator Windows 7 Charter Member
- MCITP - Microsoft Certified IT Professional Enterprise Support Technical
- MCPD - Microsoft Certified Professional Developer: Web Applications
- MCTS - Microsoft Certified Technology Specialist: Windows 7 Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows Mobile 6. Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Active Directory Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Networking Charter Member
- MCTS - Microsoft Certified Technology Specialist: System Center Configuration Manager
- MCTS - Microsoft Certified Technology Specialist: System Center Operations Manager
- MCTS - Microsoft Certified Technology Specialist: Exchange 2007
- MCTS - Microsoft Certified Technology Specialist: Windows Sharepoint Services 3.0
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2008
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 3.5, ASP.NET Applications
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2005
- MCTS - Microsoft Certified Technology Specialist: Windows Vista
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 2.0
- MCDBA – Microsoft Certified Database Administrator (SQL Server 2000/OLAP/BI)
- MCAD – Microsoft Certified Application Developer .NET
- MCSA 2000 – Microsoft Certified System Administrator Windows 2000
- MCSA 2003 – Microsoft Certified System Administrator Windows 2003
- Microsoft Small and Medium Business Specialist
- MCP – Visual Basic e ASP
- MCT – Microsoft Certified Trainer
- SUN Java Trainer – Java Core Trainer Approved
- IBM Certified System Administrator – Lotus Domino 6.0/6.5

1	Introdução	4
1.1	Metodologias Para Desenvolvimento de Soluções	4
1.2	Introdução a MSF / RUP	4
1.2.1	Modelo de Times	5
1.2.2	Modelo de Processos	6
1.2.3	Gerenciamento de Riscos	7
1.2.4	Gerenciamento de Projetos	7
1.2.5	Documentação	8
2	Visão / Escopo	9
2.1	Introdução	9
2.2	Técnicas de Análise	9
2.2.1	Identificação de Escopo	10
2.3	Use Cases e Cenários	10
2.4	Refinamento e Fechamento	13
3	Planejamento – Modelo Conceitual	14
3.1	Introdução a Fase de Planejamento	14
3.2	Pesquisa	14
3.3	Análise	14
3.3.1	Arquitetura da Aplicação	15
3.4	Otimização	16
4	Planejamento – Modelo Lógico	16
4.1	Introdução	16
4.2	Análise	17
4.2.1	Tecnologias Candidatas	17
4.2.2	Definição de Atributos e Propriedades	17
4.2.3	Definição de Multiplicidade e Relacionamento	18
4.2.4	Diagrama de Seqüência	19
4.2.5	Diagrama de Dados	20
4.2.6	Design de Alto Nível	21
4.3	Otimização	22
5	Planejamento – Modelo Físico	23
5.1	Introdução	23
5.2	Pesquisa	23
5.3	Análise	24
5.3.1	Diagramas de Classe e Seqüência Revisados	24
5.3.2	Diagrama de Database Revisada	25
5.3.3	Diagramas de Topologia e Componentização	26
5.4	Racionalização	27

5.5	Modelo de Implementação e Programação	29
5.6	Otimização	30
6	Desenvolvimento	30
6.1	Introdução	30
6.2	Provas de Conceito	31
6.3	Versões Internas (internal releases)	31
7	Estabilização / Testes	32
7.1	Introdução	32
7.2	Convergência de Bugs e Zero Bounce	32
7.3	Versões Candidatas	33
8	Entrega	33
8.1	Introdução	34
8.2	Implantação de Tecnologias	34
8.3	Transferência de Comando	34

1 Introdução

1.1 Metodologias Para Desenvolvimento de Soluções

Desde o início dos processos de desenvolvimento de software, ainda com os Mainframes, já existiam metodologias para análise e desenvolvimento de software.

Estas metodologias não especificam a tecnologia e sim modelos administrativos, documentais e organizacionais. Abrangendo desde a concepção até a entrega final da solução, metodologias baseadas no modelo UML hoje se tornaram o padrão entre as empresas e profissionais de informática.

Até a década de 80 modelos baseados em linguagens eram os mais utilizados o que muitas vezes dificultava a migração de uma aplicação entre diferentes linguagens e sistemas operacionais. Em meados dos anos 90, três grandes “pensadores” da chamada engenharia de software, Jacobson, Booch e Rumbaugh, criaram o modelo *Unified Modeling Language* que conhecemos e nos referimos com o UML. Este padrão de engenharia foi amplamente aceito por utilizar notação gráfica e modelos de processos completos e versáteis.

Inicialmente a empresa Rational (hoje parte da IBM) criou junto com os três criadores do UML a ferramenta chamada de Rose. Com a facilidade de um modelo para soluções padronizado e uma ferramenta visual bem elaborada, empresas como Digital, HP, Unisys, IBM, Microsoft, Xerox entre outras, adotaram o UML em seus processos internos de desenvolvimento. Hoje as principais ferramentas UML são o XDE da Rational, Visio da Microsoft e Together da Borland.

Mas engana-se quem ainda segue o princípio “Ivonsaf” (Irresistível VONTade de SAir Fazendo) de que apenas conhecer UML e criando os modelos gráficos por ele definido o software será um sucesso. É necessário conhecer todo o processo envolvido na criação da solução proposta. Uma pesquisa realizada em 2000 pela Standish Group International em entrevista a mais de 30.000 profissionais de TI resultou na estatística abaixo:

Falharam (23%)	Desafiadores (49%)	Sucesso (28%)
----------------	--------------------	---------------

Porque isto acontece?

Os motivos listados foram falta de comunicação dentro da equipe, má interpretação das necessidades do usuário, falta de recursos (dinheiro, pessoal e equipamentos), foco em tecnologia e não nos negócios e processos inflexíveis.

Como o UML apenas abrange uma notação padronizada para as soluções, surgiram metodologias que complementassem esta notação por incluir gerenciamento de processos, riscos, documentação e equipes. Como exemplo temos o modelo RUP (Rational Unified Process), o MSF (Microsoft Solutions Framework) e o PMI (Project Management Institute).

1.2 Introdução a MSF / RUP

O Microsoft Solutions Framework é baseado em UML e no RUP da Rational, lançado como produto em 93. Seu surgimento foi ocasionado por um levantamento interno realizado por ex-funcionários da Anderson Consulting que haviam sido contratados pela Microsoft para unificarem os processos de desenvolvimentos dos diversos times existentes. Notou-se que UML era utilizado por todas as equipes e que os modelos administrativos e gerenciais eram similares aos utilizados pela Rational (RUP). Documentando estes processos, o MSF nos permite controlar e racionalizar uma solução. Utilizamos o MSF como base por ser o modelo utilizado em todo o processo de desenvolvimento com .NET.

Atualmente o modelo MSF está na versão 3.0 e é formado por cinco principais conceitos:

- Modelo de Times
- Modelo de Processo
- Gerenciamento de Riscos
- Gerenciamento de Projetos
- Documentação

1.2.1 Modelo de Times

Os envolvidos no desenvolvimento da solução foram separados em 6 times:

- Gerente de Produto (Product Manager)
É o responsável pela comunicação com o usuário detentor dos recursos (Stakeholder). Este profissional possui um conhecimento básico de informática, uma vez que a sua principal função será conhecer o cliente e “vender” o produto.
- Gerente de Projeto (Project Manager)
Gerencia recursos como dinheiro, pessoal, equipamentos e prazos do projeto. Não é o chefe nem o encarregado técnico do projeto, portanto seu conhecimento tecnológico precisa ser amplo e não específico. Faz parte de sua equipe os analistas de sistemas e engenheiros de software. A estes cabe definir as tecnologias, mas não implantá-las.
- Desenvolvedores (Developers)
São os técnicos envolvidos no projeto, desde o programador até o pessoal de segurança e sistemas operacionais. Estes possuem alto nível técnico especializado pois serão os responsáveis pela implementação, implantação e codificação.
- Testes (Testers)
Equipe que irá conduzir os testes da solução. Em alguns tipos de testes, por exemplo de segurança, necessitam ter alto nível técnico, enquanto em outros testes, por exemplo testes de navegação, o nível técnico exigido é menor.
- Usabilidades (User Testers)
Ocupam um importante papel por serem conhecedores da solução e não de tecnologia. Caberá a estes avaliar se as necessidades e expectativas do cliente e usuários finais foram atingidas.
- Gerencia de Produção/Operação (Release Manager)
Equipe de suporte e operação, responsáveis pela implantação e suporte quando a solução estiver em produção.

A vantagem do modelo de equipes é óbvia por definir responsabilidades o que evita conflitos. O modelo de equipe também prega os seguintes conceitos:

- Visão compartilhada
Cada membro da equipe precisa conhecer o objetivo do projeto, o seu papel e os prazos envolvidos. Quando os membros da equipe não se sentem partes do projeto perdem a iniciativa, o que ocasiona perda de qualidade.
- Foco em Negócios
A maior parte dos processos que fracassam tiveram seu foco em tecnologia e não no negócio. Projetos que visam “explorar” uma nova tecnologia ou é visto como oportunidade para aprender um novo produto, tende a estar entre os 77% dos projetos ineficientes. Tenha em mente que trabalhamos em organizações comerciais que visam lucro e não para as empresas de telecomunicações e software. Isto exige alta disciplina, mas resulta em projetos lucrativos.

- **Ágil e flexível**
Nos anos em que os Mainframes e CPDs eram reinantes, a mudança de uma pequena frase em um documento gerado pelo sistema provocava uma guerra entre técnicos e usuários. Hoje em dia a realidade é totalmente diferente, o poder de processamento dobra a cada 18 meses (regra de Moore) e as organizações se reestruturam a cada três anos, criando mudanças significativas nos processos e sistemas de uma empresa.
- **Comunicação**
“Uma equipe que tem medo de se expressar é uma equipe de zumbis”. Esta frase define bem o que acontece quando os gerentes e líderes não permitem a liberdade de pensamento. Todos os integrantes da equipe precisam se sentir à vontade ao dar opiniões, sem medo de serem punidos, repreendidos ou ridicularizados. Fornecer meios de ouvir os membros da equipe, mesmo que anonimamente, aumenta a eficiência por ouvir e motivar a base de produção. Empresas como Alcoa, Itaú, Randon e muitas outras montaram sistemas de premiação a boas idéias.

1.2.2 Modelo de Processos

Dividir o projeto em fases é utilizado nos modelos MSF 3.0, PMI e RUP. Apesar da separação em fases, todo o modelo é baseado em “Spiral” e “Waterfall”.

O modelo Espiral define que o projeto pode ser particionado, e que cada funcionalidade do projeto pode ter seu desenvolvimento iniciado assim que suas documentações estiverem prontas. Por exemplo, em um sistema para locação de imóveis podemos dividir o projeto em cadastro de imóveis, cadastro de clientes e vendas. Ao terminar a documentação do cadastro de imóveis já iniciamos o desenvolvimento enquanto documentamos o cadastro de clientes.

O modelo Waterfall é baseado em “millestones” ou marcos. Estes marcos definem onde cada uma das cinco fases principais ou suas sub-fases começam e terminam. Os marcos tem como atrativo deixar as equipes sincronizadas

Com a junção dos dois modelos podemos modularizar um projeto (espiral), definindo quando cada módulo é considerado documentado (marco) para o início do desenvolvimento.

As cinco fases do processo são:

- **Visão (Envisioning)**
Nesta fase conduzimos as entrevistas e criamos os use-cases que são as definições do que a solução fará e o que não fará. A isto chamamos de Escopo (scope).
- **Planejamento (Planning)**
Dividida em três sub-fases, todas as documentações do sistema são geradas nesta fase. É considerada uma fase crítica, pois um erro de planejamento pode resultar na reescrita de tudo o que já estiver pronto.
- **Desenvolvimento (Developing)**
Inclui desde a codificação até testes com sistemas operacionais e tecnologias. Tipicamente a maior fase do projeto.
- **Estabilização (Estabilizing)**
Testes com usuários internos e externos (beta users) são conduzidos de forma sistemática até conseguir se chegar ao chamado Golden Release, versão final do produto para venda ou implantação.
- **Entrega (Deployment)**
Última fase mas não menos importante, é quando criamos a política e efetiva implantação do

projeto. Nesta fase se torna difícil qualquer correção no sistema, uma vez que as outras equipes já foram desmontadas.

1.2.3 Gerenciamento de Riscos

O que é um risco?

Riscos são quaisquer fatores que possam causar dificuldades. Os riscos não se resumem a tecnologia. Podem ser organizacionais, administrativos, financeiros, pessoais, etc.

Qualquer projeto, desde o menor até os faraônicos, possui uma lista de fatores que poderão naufragar o projeto. Alguns projetos possuem poucos riscos, mas a maior parte possui tantos riscos que precisamos formular quais são mais importantes.

Para categorizar riscos precisamos da participação de todos os membros de todas as equipes, uma vez que para o gerente de produto os riscos envolverão o cliente, para os desenvolvedores riscos técnicos e para a equipe de usabilidade os costumes do usuário.

Utilizando um modelo básico de riscos, veja o exemplo abaixo:

Area	Risco	Probabilidade	Impacto	Exposição	Mitigação	Contingência
G.Proj	Orçamento enxuto, prazo curto	4	5	20	Contratar profissionais com bom gabarito e skill técnico	Terceirização de técnicos na implantação
Desenv	Tecnologia é recente, documentação escassa	3	3	9	Antes da escolha do wi-fi verificar documentações disponíveis	Contratação de um profissional experiente na tecnologia
G.Prod	Difícil acesso ao dono da empresa	4	2	8	Marcar as reuniões e apresentações do sistema com antecedência	Definir um substituto com poder de decisão equivalente

Neste exemplo notamos as colunas probabilidade que indica a espera do risco, o impacto caso a situação se confirme e a exposição que é o resultado da multiplicação dos dois primeiros. Esta coluna é essencial para definição dos “Top 10”.

Mitigação são recursos que podem ser usados previamente a fim de evitar a situação, enquanto contingência é um recurso que não queremos utilizar por qualquer motivo que seja, mas terá que ser utilizada em ultimo caso. Nesta planilha não consta a coluna “Disparo”, permitindo definir quando a contingência será necessária.

1.2.4 Gerenciamento de Projetos

Projetos possuem três principais sustentadores, chamados de tríade:

- Recursos
Itens necessários para o projeto, destacando-se pessoal, dinheiro e equipamentos.
- Prazos
Prazos máximo do projeto e de cada uma das suas fases
- Funcionalidades

As tarefas que o usuário definiu como sendo parte do sistema

O bom gerenciamento da tríade é conseguir definir qual das três é mais importante, qual é aceitável e, por conseguinte, a que é ajustável. Veja os exemplos abaixo:

	Fixo	Aceitável	Ajustável
Prazo Não pode ser alterado, entrega até dia 01/10	✓		

Recursos Equipe já existente com 5 pessoas. Não temos autorização para aumento da equipe		✓	
Funcionalidade Os módulos 1 a 5 são prioritários. Os módulos restantes poderão ser desenvolvidos em outra ocasião			✓

O exemplo acima demonstra que o prazo é inegociável, recursos foram aceitos como estão e que as funcionalidades estão comprometidas. Fica claro ao cliente que para fazer no prazo especificado com o pessoal fornecido não nos comprometemos com todas as funcionalidades desejadas. Cabe ao gerente de projeto definir um cronograma com todos os marcos e com este cronograma em mãos cuidar de que os prazos não sejam ultrapassados. O mesmo deve ser feito utilizando uma lista de equipamentos e pessoas para controle.

1.2.5 Documentação

A documentação é essencial para o desenvolvimento e também para o cliente. Todos os papéis gerados durante o planejamento deve ser anexado ao documento inicial de Visão/Escopo formando uma pasta para referência.

Também durante o projeto são gerados documentos de suporte a usuário, chamados internacionalmente de KB (Knowledge Base), bem como manuais, resolução de problemas comuns, etc. Um sistema bem documentado também incluirá padrões de desenvolvimento e codificação, por exemplo, nomenclatura de objetos, database e servidores.

NOTA: Os documentos gerados em um projeto serão demonstrados ao longo do curso.

2 Visão / Escopo

2.1 Introdução

O início de um projeto é marcado por um “financiador” (stakeholder) ou cliente que ao ter uma idéia sugere a criação do sistema. A partir deste momento passamos a ter o envolvimento de pelo menos um representante de cada equipes para definição do documento de visão.

Este documento define o escopo do projeto, que indica ao cliente as funções que serão implementadas e as que não serão. Se ao definir um escopo prometermos mais do que podemos fazer com o dinheiro e prazos definidos, o projeto irá naufragar. Não é necessário neste momento definir tecnologias, mas será necessário pensar nelas, afinal, não podemos prometer algo que a atual tecnologia ou os custos impossibilitem.

2.2 Técnicas de Análise

São diversos os métodos de análise que podemos utilizar, mas precisamos ter cuidado com o tipo de público, nível de responsabilidade e nível hierárquico. Veja abaixo alguns exemplos:

Método	Público	Vantagens	Desvantagens
Shadow (sombra) Seguir o usuário final, anotando atividades do seu dia-a-dia	-Baixo conhecimento -Baixo nível de decisão	Fornecer dados importantes de como a tarefa é executada, quem a executa, em que ordem e quando.	Raramente consegue-se saber porque determinadas tarefas são executadas.
Entrevista Método comumente utilizado por uma conversa informal	-Alto conhecimento -Alto nível de decisão	Permite conhecer os motivos que levam ao processo, quem o executa e porque, bem como detalhes importantes sobre a empresa e o negócio.	Demanda muito tempo, muitos detalhes ficam ocultos. Não conseguimos enxergar dificuldades operacionais.
Grupos (JADE) Reunir um grupo de pessoas que estejam envolvidas	-Médio conhecimento -Médão nível de decisão	Levanta-se o processo dos diferentes pontos de vista dentro da organização.	Demanda muito tempo, e muitas vezes desenvolve-se como uma discussão entre áreas, onde a mais forte tenta predominar.
Pesquisa Elaborar questionários a serem preenchidos	-Médio conhecimento -Baixo nível de decisão	Fornecer um amplo ponto de vista, é anônimo e não compromete cargos de alto nível, não tomando tempo desnecessário.	Divagação freqüente, processos mal-descritos, tempo de retorno dos questionários altamente variável, falta de confiabilidade.
Manuais (outputs) Utiliza-se o manual de operações, retornos de dados ou impressões	Não se aplica	Dados em tempo real e oficiais da empresa, altamente confiáveis. É o melhor método quando da migração de sistemas legado.	Acomodação nos mesmos moldes, falta de criatividade e solução que apenas ficou mais “atraente”.
Protótipo Criar um modelo para ser testado por técnicos ou usuários	Não se aplica	Quando é um novo projeto na organização, permite avaliar aceitação e funcionamento.	Exige que um sistema Delta seja criado, o que pode demandar muito tempo. Informações desencontradas em cada nível de usuário.

2.2.1 Identificação de Escopo

Após as análises estarem prontas, precisa-se definir o escopo, ou os requisitos do sistema. Esses podem ser divididos basicamente em quatro tipos:

- **Funcionais**
Definem o que o sistema deverá fazer, como fazer e quando o faz.
Por exemplo, o sistema deverá ser capaz de registrar os produtos, estoque e as vendas, bem como gerar históricos e relatórios.
- **Dados**
Define a estrutura estática, os dados que deverão ser guardados.
Por exemplo, o produto deverá conter código, descrição, preço, estoque mínimo e estoque atual.
- **Interface**
Envolve o usuário na operação, pode ser uma tela de input ou um relatório.
Por exemplo, a recepcionista deve preencher o RG, nome e pessoa visitada, digitando logo em seguida o número do crachá fornecido.
- **Não-funcional**
Identifica partes da solução que não são diretamente ligadas a ela. Podem ser requisitos de segurança, sistema operacional, meio de acesso, etc.
Por exemplo, o sistema deverá ser desenvolvido em C# executado no Windows 2003 e com chave de criptografia de 128 bits modelo RSA.

2.3 Use Cases e Cenários

Use cases, ou casos de uso, identificam a ordem em que os eventos acontecem, quem o executa, quem ou o que está envolvido e como será executado.

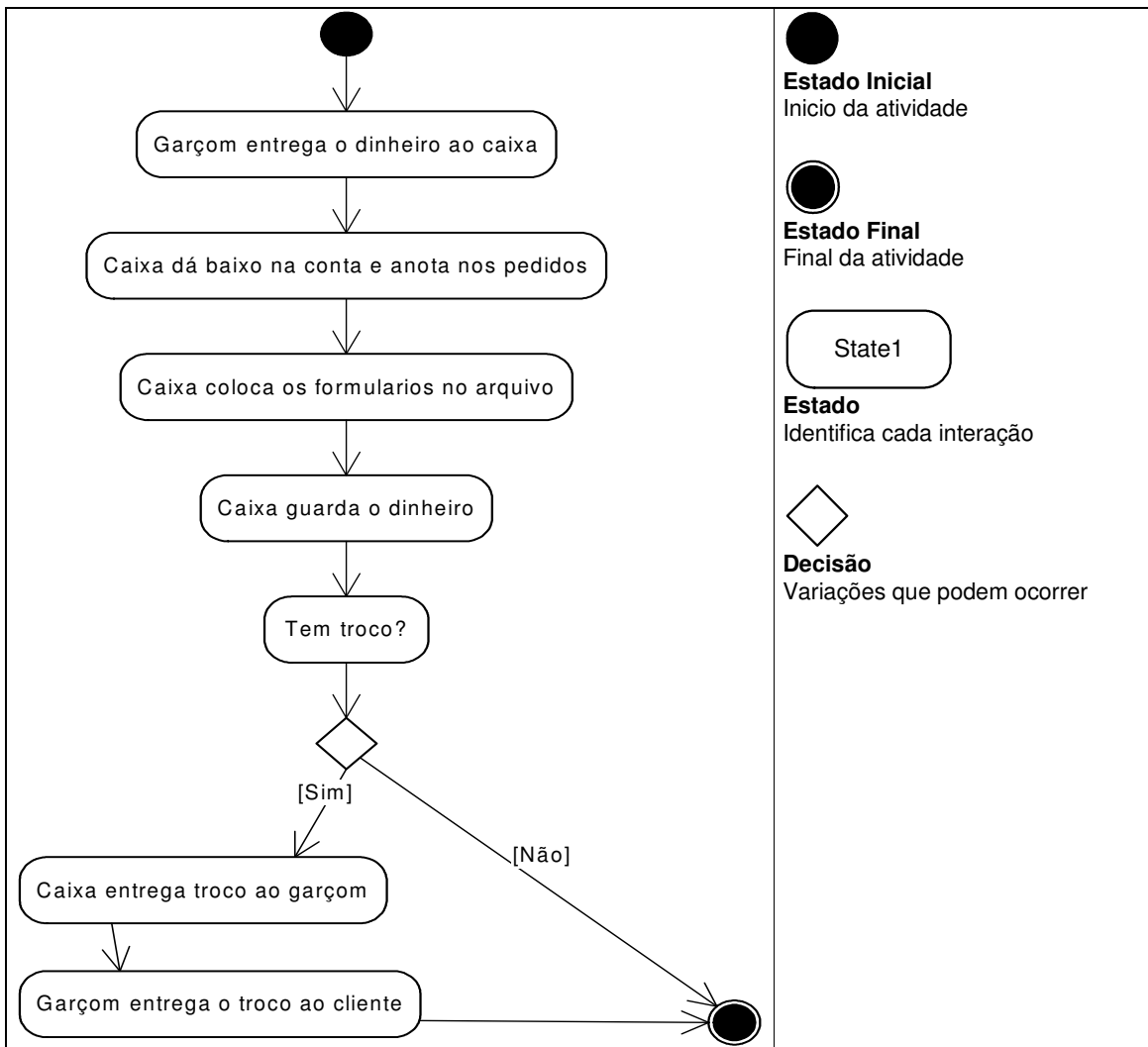
Os UC surgem quando se refina as entrevistas tornando-as lógicas. Um UC define funcionalidades do sistema, enquanto os cenários definem variações desta funcionalidade.

Por exemplo, imaginando um sistema de controle em um restaurante podemos definir o primeiro UC como sendo o atendimento do garçon:

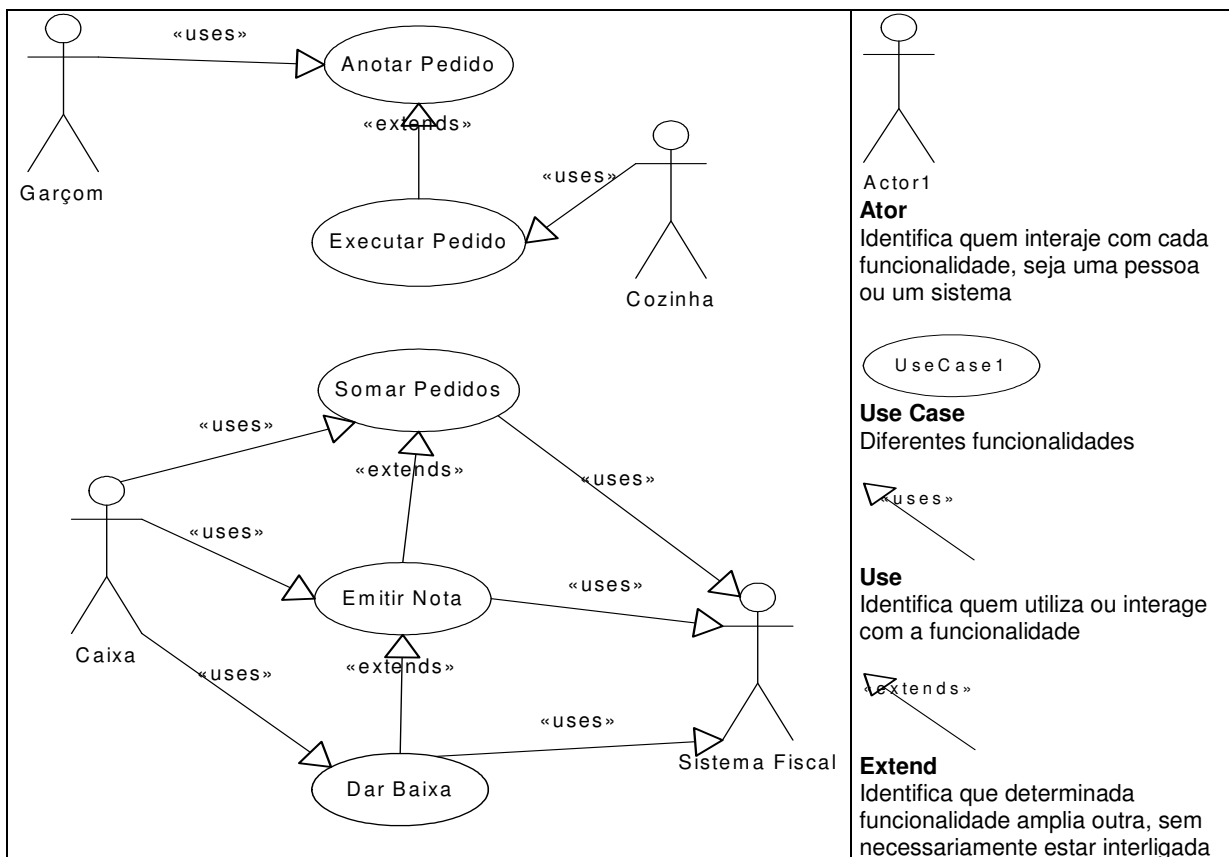
Cenário	Tarefas envolvidas
UC 1.1 Cliente solicita produto	<ol style="list-style-type: none"> 1. Garçon anota o pedido 2. Destaca a via da cozinha e leva ao escaninho 3. Ajudante da cozinha pega o formulário 4. Cozinha prepara o pedido 5. Garçon entrega o pedido na mesa 6. Cozinha entrega o formulário ao caixa 7. Caixa coloca o formulário na colméia
UC 1.2 Cliente solicita a conta	<ol style="list-style-type: none"> 1. Garçon passa o numero da mesa ao caixa 2. Caixa retira os formulários da colméia 3. Soma os pedidos 4. Digita os dados no sistema fiscal 5. Imprime a conta 6. Garçon leva a conta ao cliente
UC 1.3 Cliente paga em dinheiro	<ol style="list-style-type: none"> 1. Garçon entrega o dinheiro ao caixa 2. Caixa dá baixa na conta e anota nos pedidos 3. Caixa coloca os pedidos no escaninho de arquivo 4. Caixa guarda o dinheiro 5. Se houver troco o caixa entrega ao garçon 6. Garçon entrega o troco ao cliente, se houver

Como pode ser visto neste exemplo, criamos um UC com três diferentes cenários.

Também é possível definir os cenários de forma gráfica. A primeira delas é o diagrama de atividade, também chamado de fluxograma:



Os use cases também podem ser representados graficamente, mas neste caso eles exemplificam a relação entre os diferentes módulos e interações entre o sistema:



Este diagrama mostra como utilizar os UCs gráficos, e como estes podem servir mais tarde para identificar as diferentes classes que o sistema vai conter. Veremos no módulo seguinte que os UCs serão refinados na etapa de Planejamento. Vamos entender melhor como utilizar os UCs gráficos. Ao ler uma entrevista, um UCs descritivo ou um diagrama de atividades, identifique os atores, objetos e ações especificadas. Atores podem ser facilmente identificados porque manipulam os objetos. Objetos são os itens que foram manipulados. Por fim, ações são verbos, as manipulações. Veja os exemplos abaixo:

Atividade	Ator	Objeto	Ação
Garçom anota o pedido	Garçom	Pedido	Anotar
Caixa lança pedidos no sistema	Caixa Sistema	Pedido	Lançar
Cozinha executa o pedido	Cozinha	Pedido	Executar

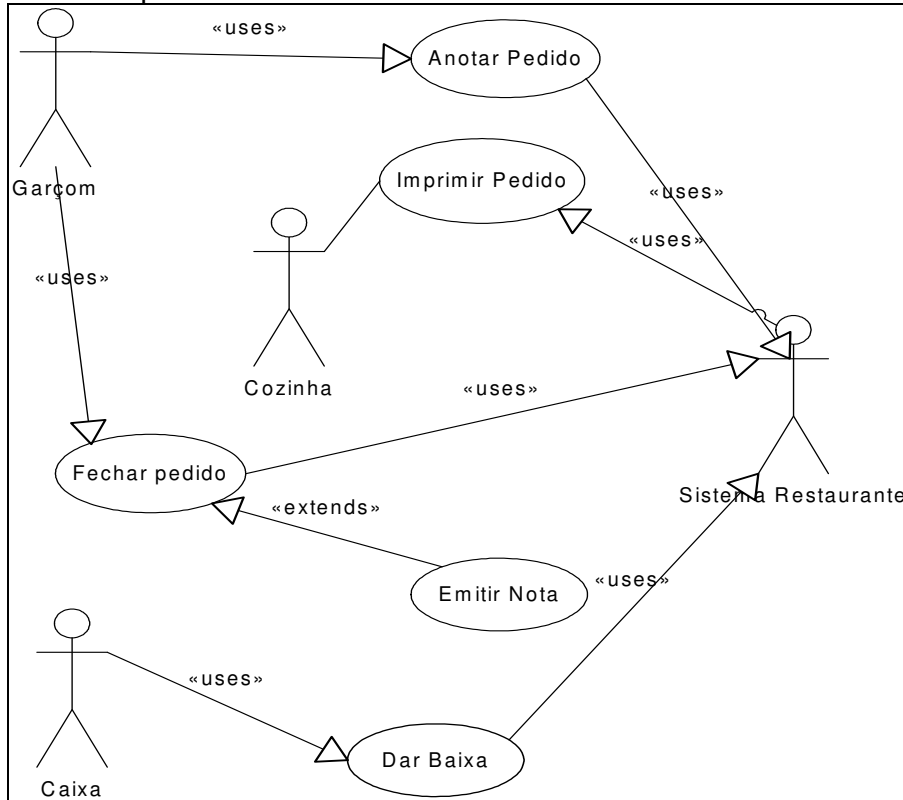
Esta lista ainda não é definitiva, ela será revista na fase de planejamento. Chamamos esta lista de Candidatos.

Alem do UML outro modelo muito utilizado para definir UCs é chamado de Modelagem dos Papeis dos Objetos (ORM – Object Role Modeling). O ORM descreve os UCs de forma analítica, permitindo um entendimento dos cases por não utilizar desenhos técnicos como o UML. Por outro lado, o ORM gera muito mais papeis e documentações do que o UML, pois com um gráfico de UCs e mais três gráficos de atividade que podem estar em uma única folha, conseguimos representar as três operações analisadas. Com ORM teriamos um modelo descritivo em linguagem natural. Outra solução muito adotada é utilizar o ORM inicialmente para descrever as atividades e mais tarde fazer os diagramas gráficos de UML, já como refinamento.

2.4 Refinamento e Fechamento

Refinamento é quando pegamos os UCs e utilizando o método Walk-Throggle (andar sobre ele) verificamos se as informações estão consistentes. Isto é feito por se colocar no lugar de um usuário e seguindo os diagramas de atividade simular a tarefa sendo executada. Com este testes podemos identificar os chamados “objetos ocultos”. Com a descoberta destes objetos, refazemos os modelos. Outra importante função do refinamento é conseguir mapear o “Estado Atual” e o “Estado Futuro”. Estes consistem em descrever como atualmente o processo é executado e o segundo modelo como será executado com o novo sistema.

Um exemplo do UCs citado anteriormente e o estado futuro seria:



Cenário	Tarefas envolvidas
UC 1.1 Cliente solicita produto	<ol style="list-style-type: none"> 1. Garçom anota o pedido no Pocket PC 2. A impressora da cozinha automaticamente emite um ticket 3. Ajudante da cozinha pega o ticket 4. Cozinha prepara o pedido 5. Garçom entrega o pedido na mesa
UC 1.2 Cliente solicita a conta	<ol style="list-style-type: none"> 1. Garçom digita o fechamento no Pocket PC 2. Garçom recebe o valor do cliente 3. A impressora do caixa emite o recibo 4. Garçom entrega o dinheiro ao caixa 5. Caixa da baixa no sistema 6. Caixa guarda o dinheiro 7. Se houver troco o caixa entrega ao garçom 8. Garçom entrega o troco ao cliente

3 Planejamento – Modelo Conceitual

3.1 Introdução a Fase de Planejamento

A fase de planejamento se destaca em um projeto por ser considerada o cerne de uma solução. Imagine um engenheiro desenhando um edifício de 6 andares e esquecer de colocar as tubulações de telefone. Este engenheiro pode contar com a sorte de ter um mestre de obras consciente que lhe avise da falta e seja corrigido, mas caso isto não aconteça os moradores irão ficar insatisfeitos e com certeza resultará em prejuízos a construtora.

Similarmente, um projeto de sistema tem a mesma fragilidade. Se a fase de planejamento for mal feita, provavelmente os programadores não enxergaram isto por não conhecerem diretamente o usuário final. Quando o cliente receber o sistema pronto, este notará dificuldades e assim como aquele morador insatisfeito, assim ficará seu cliente.

A fase planejamento como já abordado anteriormente, é dividida em três fases (conceitual, lógica e física) e formulada de forma espiral, ou seja, dividimos o projeto em porções, os use cases, e executamos o planejamento e desenvolvimento conforme o fechamento de cada use case.

3.2 Pesquisa

Nesta etapa os analistas de sistemas com um nível técnico mais alto e que não necessariamente estiveram envolvidos na fase de visão, quando as entrevistas foram realizadas, irão analisar os UCs e fazer o processo chamado de “restate” ou re-estado.

Isto significa analisar as atividades de forma sistêmica, ou seja, como um programa de computador, uma vez que até este momento os use cases e cenários foram sempre analisados do ponto de vista do usuário.

Com este método de pesquisa será possível contestar, sugerir inclusões ou exclusões de atividades. Utilizamos o termo “sugerir” porque ao propor alterações já com os UCs montados precisamos que os analistas que originalmente os montaram aprove as alterações e que o cliente também o faça, uma vez que no final da fase de visão e escopo foi obtido um aval por parte do cliente. Estas alterações caso impliquem em extensão do prazo ou dos gastos tem que ser aprovada novamente. Nesta fase também é levantado o User Profile, ou perfil do usuário, que é a identidade técnica e educacional de quem irá estar operando o sistema. Por exemplo, se o perfil for escritório podemos desenvolver interfaces mais ricas, enquanto o perfil fabril exige interfaces simples, muitas vezes sem uso de mouse.

3.3 Análise

Na fase de análise dos use cases iremos identificar os objetos principais, atores e ações criando um documento que sintetize o que cada ator manipula, entender requisitos técnicos como por exemplo, segurança, escalabilidade, etc.

Exemplificando um modelo re-analisado e sintetizados temos:

Ator	Atividades
Garçom	<ul style="list-style-type: none">• Logar no Pocket PC• Cadastrar itens na mesa• Consultar a conta da mesa• Fechar a conta da mesa• Informar a forma de pagamento
Cozinha	<ul style="list-style-type: none">• Imprimir pedido
Caixa	<ul style="list-style-type: none">• Logar no sistema

	<ul style="list-style-type: none"> • Consultar a conta da mesa • Imprimir o cupom fiscal • Processar a baixa do pedido
--	---

Como visto neste documento, a função “Logar no sistema” e “Informar a forma de pagamento” não estavam listadas entre as atividades anteriormente levantadas nos use cases originais.

Na análise conceitual do sistema concentra-se algumas das mais críticas decisões do projeto, que envolvem a escolha da arquitetura da aplicação.

3.3.1 Arquitetura da Aplicação

Para entender o que representa a arquitetura da aplicação precisamos entender os modelos de desenvolvimento em camadas e serviços.

Desenvolvimento em serviços envolve separar os diferentes atores, objetos e ações conforme a função que desempenham. Os serviços são:

- **Apresentação**
Estes envolvem a UI (User Interface) incluindo formulários, relatórios, páginas web e quaisquer outras formas de entrada de dados.
- **Negócios**
Encapsulam, ou controlam, as regras do sistema.
- **Dados**
Define basicamente a persistência de dados, mas muitas vezes pode representar dados dinâmicos e temporários.

A planilha abaixo demonstra como o sistema visto anteriormente seria classificado:

Componente	Função	Apresentação	Negócios	Dados
Garçom	Dados dos garçons (login)			✓
Caixa	Dados dos caixas (login)			✓
Login	Tela de logon	✓		
ValidaLogin	Verifica se os dados estão corretos		✓	
Vendas	Tela de inserção para os pedidos	✓		
Produtos	Dados de produtos para venda			✓
Pedidos	Dados dos pedidos			✓
Cozinha	Imprime os pedidos na cozinha	✓		
Conta	Soma os pedidos		✓	
Recebimento	Cadastra as formas de pagamento			✓
Recibo	Imprime o recibo	✓		

Este exemplo demonstra dois conceitos importantes. O primeiro notamos ao ver que a lista de componentes inclui aqueles refinados anteriormente e que não constavam no UC por serem “ocultos”, como por exemplo, o componente recebimento e conta. O segundo conceito é a separação de serviços. Os componentes de apresentação se transformaram em formulários, páginas web e relatórios. Quanto aos componentes de negócios serão implementados como DLLs e os componentes de dados se tornam tabelas em banco de dados, XML ou outro sistema de persistência.

Como pode ser notado, os serviços são importantes para definir em que linguagens, sistema ou servidor será implementado cada uma das diferentes partes que compõe a solução.

Agora vamos entender o modelo de camadas. Uma aplicação é chamada de Monolítica quando seus diferentes serviços estão todos vinculados a um único assembleie (programa que pode ser dll, exe, com, etc.). Por exemplo, imagine o programa desenvolvido pela Receita Federal para elaboração do imposto de renda, desenvolvido em Delphi em um único executável, para facilitar a distribuição. O

fato de um sistema ser monolítico não quer dizer que não tenha o conceito de serviços, mas sim que ele foi compilado como uma única parte. Podemos desenvolver sistema baseados em serviços mas implementado em um único assemble, como na situação da Receita Federal.

Quando um sistema desenvolvido no modelo de serviços é implementado em diferentes assemblies chamamos este sistema de multi-tier ou layered (multicamadas). Outro modelo muito utilizado é o modelo Client-Server (cliente-servidor) onde temos a camada de apresentação em um programa desenvolvido em Visual Basic, por exemplo, enquanto os dados estão em um banco de dados relacional como o MS-SQL Server ou Oracle.

A escolha do modelo depende muito da aplicação e está vinculado a alguns fatores:

Modelo	Vantagens	Desvantagens
Client-Server O software cliente é instalado na maquina do cliente ou em um servidor compartilhado e os dados em um servidor DBMS.	-Acesso a dados em uma única fonte, dados centralizados -Baixo processamento do cliente -Fácil distribuição -Fácil desenvolvimento	-Alta conectividade -Em caso de falta de comunicação todo a solução irá parar -Difícil manutenção quando instalado nos clientes -Performance comprometida
Multi-Tier / Layered Cada diferente parte do software está separada. Os serviços de apresentação no cliente ou servidor compartilhado, os de negócios em um servidor de aplicação e os dados em um servidor DBMS.	-Cada componente pode ser manipulado individualmente -Alta escalabilidade, por permitir que cada servidor leve sua própria carga -Manutenção simplificada	-Difícil distribuição, diferente em cada camada -Complexo desenvolvimento -Alta conectividade -Performance comprometida
Stateless (Estado Cheio) Ao se conectar a aplicação o cliente recebe todos os dados e devolve ao finalizar as alterações que houver efetuado.	-O cliente só necessita de conexão no inicio e fim das operações -Alta escalabilidade, pois os clientes não ficam conectados -Alta performance	-O cliente não enxerga atualizações real-time por outros clientes, ocorrendo conflitos constantes -Complexo desenvolvimento
Layered-Client-Server-Stateless Junção dos modelos, onde o cliente se conecta pede os dados necessários na função que irá executar, envia atualizações e atualiza os dados localmente.	-Alta escalabilidade -Acesso a dados centralizado -Alta performance	-Complexo desenvolvimento -Alta conectividade

3.4 Otimização

Neste ultimo esforço do planejamento conceitual é quando validamos tudo o que foi analisado e pesquisado. Faça o caminho do usuário (walk-through) e verifique se compriu com os requisitos exigidos. Uma série de perguntas pode ser feita para ajudar:

- O processo se tornou mais simples ?
- Está em conformidade com o perfil do usuário ?
- Melhorou a performance nas atividades ?
- Consigo apontar eficientemente ROI (retorno de investimento) ?
- Eliminei papeis e burocracias ?

Estes exemplos são úteis para ilustrar as diferentes questões que mostram se a sua solução é ou não viável como está sendo proposta e se os recursos empregados irão valer a pena.

4 Planejamento – Modelo Lógico

4.1 Introdução

No modelo lógico desenvolvemos a aplicação de forma prática. É nesta fase, tão critica quanto a fase conceitual, que definimos bases de dados, classes e modelos. Não é agora que montamos as

tabelas em um DBMS, nem codificamos ou definimos a tecnologia física, mas criamos os parâmetros técnicos necessários. Uma boa classificação para o lógico é dizer que serve de ponte entre o modelo conceitual (ponto de vista do usuário) para o modelo físico (ponto de vista dos técnicos).

4.2 Análise

Na fase de análise iremos definir tecnologias candidatas, objetos, atributos, relacionamentos e sequencia de operações. Esta fase é a única da fase lógica, seguida da otimização, assim como no modelo conceitual.

4.2.1 Tecnologias Candidatas

Iniciamos a fase lógica por escolher as tecnologias candidatas. Estas podem ser diversas, mas ao final do modelo físico serão nomeadas. Alguns dos fatores são:

Tipo	Requisito	Descrição
Negócios	Possibilidade	As tecnologias existem e estão disponíveis?
	Custos	O custo é viável?
	Experiência	O mercado tem documentação e profissionais habilitados?
	ROI	O prazo de retorno de investimento é aceitável?
	Maturidade	A tecnologia está madura e estável?
	Suporte	Ao encontrar problemas, qual o tempo de solução?
Empresa	Equipamentos Atuais	Fornecem o hardware necessário?
	Equipamentos Futuros	A empresa tem os pré-requisitos para a nova tecnologia?
Tecnológicas	Segurança	A tecnologia tem criptografia, autenticação, autorização e outros recursos?
	Interoperabilidade	Consegue conversar com outros sistemas em modelos padronizados, como XML por exemplo?
	Acesso a Dados	Fornece drivers para acesso simplificados que permitam acesso rápido e simples?
	Persistência	Os dados guardados serão de rápido acesso e confiáveis?
	Serviços de Sistema	Possui suporte a transações atômicas, comunicação assíncrona e outros?
	Ferramentas RAD	As ferramentas de desenvolvimento são usuais e simples?
	Sistema Operacional	Permite criptografia de tráfego, suporte a diferentes hardwares, etc?

Um exemplo de tecnologias candidatas poderia ser:

Requisito	Candidatos
Linguagem de programação	<ul style="list-style-type: none"> • C# • VB.NET • Java
Ferramenta RAD	<ul style="list-style-type: none"> • Microsoft Visual Studio 2003 • Borland Jbuilder
Sistema Operacional	<ul style="list-style-type: none"> • Microsoft Windows Server 2003 • IBM WebSphere
Equipamentos	<ul style="list-style-type: none"> • Handheld com Pocket PC 2003 para os garçons • Palm com PalmOS 3.0 ou superior para os garçons • Pentium III com 256 MB para os caixas • Impressora matricial de 40 colunas na cozinha • Impressora térmica fiscal no caixa
Banco de Dados	<ul style="list-style-type: none"> • Microsoft SQL Server 2000 • Oracle 8 ou superior

4.2.2 Definição de Atributos e Propriedades

Atributos são tipicamente características de um objeto de dados. Por exemplo, um carro possui os atributos marca, fabricante, numero de passageiros, cavalos de força, etc.

Nesta fase pegamos os componentes anteriormente classificados e procuramos os atributos que são necessários. Normalmente serviços de apresentação não contem atributos, pois são estes componentes que recebem os dados do usuários para enviar aos componentes de negócios e dados.

Uma propriedade tem um conceito mais amplo do que atributo, pois também define ações que cada objeto executa, os chamados métodos. Estes métodos são definidos em conformidade com o que Em nosso caso iremos definir as propriedades (atributos e métodos) utilizando UML chamado de Diagrama de Classes, como exemplificado abaixo:




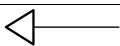
No diagrama acima já temos uma visão de que atributos e operações cada classe necessita, o que é essencial para o restante das documentações. Operações com o sinal (+) indicam públicas, o sinal (-) indica private e por fim o sinal (#) protected.

4.2.3 Definição de Multiplicidade e Relacionamento

Relacionamento indica qual componente de sua solução interage com as outras. Esta definição também inclui a multiplicidade. Multiplicidade é o numero de ocorrências incluídas de um objeto sobre o outro. Por exemplo, um pai pode ter vários (tipicamente utilizamos “n”) filhos, enquanto um filho só pode ter um pai. Usamos os seguintes nomenclaturas:

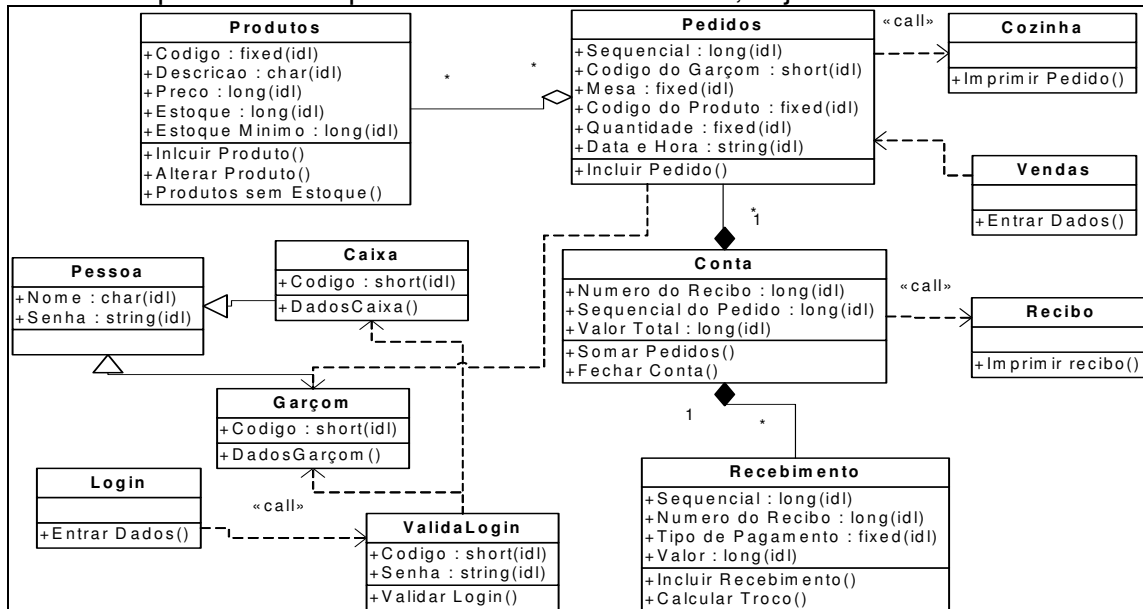
Nomenclatura	Descrição	Exemplo
1..1	Um para um	Cada pedido gera uma nota fiscal.
1..* ou 1..n	Um para muitos	Uma nota fiscal pode conter muitos produtos
..1 ou n..	Muitos para um	Diversos clientes são atendidos por um mesmo vendedor
.. ou n..n	Muitos para muitos	Diversos fornecedores podem vender diversos produtos, concorrentemente.
1..0 ou 0..1	Nenhum, ou um, para um	Cliente pode ou não ter feito pedidos até o momento

Estes valores numéricos representam a multiplicidade, mas também utilizamos alguns termos técnicos para definir relacionamentos. Por exemplo, quando o item é um para um utilizamos a expressão associação. Veja abaixo exemplos destes termos:

Nome	Símbolo UML	Exemplo
Associação	-End3 -End4  1 *	Um vendedor pode atender diversos clientes
Generalização		Podemos criar um objeto “pessoa” do qual tanto o caixa quanto o garçom tem atributos comuns.

Realização		O objeto vendas realiza um objeto pedido. No Visio utilizamos o símbolo de dependência com a descrição "call".
Dependência		O objeto login depende dos objetos garçom e caixa

Para exemplificar a multiplicidade e relacionamentos, veja o modelo abaixo:



Neste modelo é possível enxergar os relacionamentos, dependências e chamadas executadas a cada objeto, componente ou dado. Para entendermos melhor o modelo acima, vamos descrevê-lo por completo.

Objeto 1	Objeto 2	Tipo	Descrição
Pessoa	-Garçom -Caixa	Generalização	Todos os garçons e caixas são pessoas, portanto possuem atributos em comum.
Login	ValidaLogin	Realização	A tela de login chama, ou realiza, o metodo de validação do login.
ValidaLogin	-Garçom -Caixa	Dependência	Para validar um login é necessário que o usuário exista.
Pedidos	Garçom	Dependência	Um pedido só pode ser registrado caso o garçom exista.
Pedidos	Produtos	Composição	Pedidos são formados por n produtos.
Pedidos	Vendas	Realização	A tela de vendas insere os pedidos.
Pedidos	Cozinha	Realização	Ao cadastrar um pedido este automaticamente emite uma ordem para a cozinha.
Pedidos	Conta	Composição	Uma única conta contem n itens vendidos.
Conta	Recebimento	Composição	Uma única conta contem n recebimentos.
Conta	Recibo	Realização	Ao fechar uma conta automaticamente é emitido um recibo.
Produtos	Pedidos	Composição	Um mesmo produto pode ter sido vendido em n pedidos.

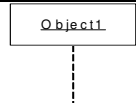

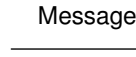
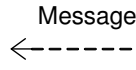
Como exemplificado acima, o modelo contem as quatro agregações bem como também os diferentes tipos de multiplicidade. Com base no modelo de agregação montamos os diagramas UML de seqüência e com o modelo de multiplicidade o diagrama de dados.

4.2.4 Diagrama de Seqüência

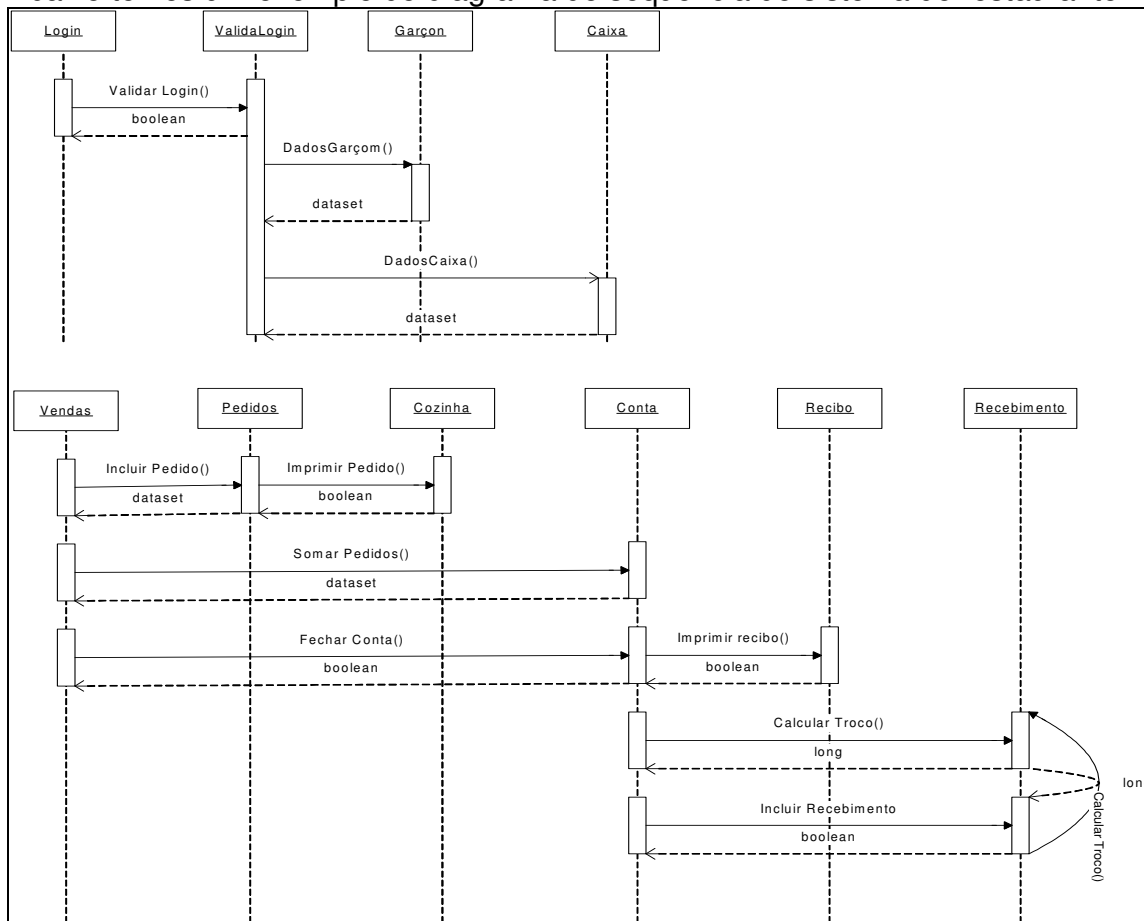
O diagrama de seqüência tem como objetivo definir a ordem de chamada dos métodos. Com este diagrama temos um controle muito simples de qual componente executa qual componente e com isto entendemos e criamos as interfaces e os retornos que cada interface deve ter. Utilizando o diagrama de seqüência podemos entender claramente a interligação e o momento de chamada.

Veja abaixo os símbolos UML utilizados nos diagramas de seqüência:

Símbolo	Descrição
---------	-----------

	<p>Objetos A linha que se estende abaixo deles é onde se coloca os símbolos de ativação. Um componente se comunica com diversos outros componentes.</p>
	<p>Ativação É colocado sobre a linha do objeto e se estende para definir todos os objetos e métodos que uma determinada chamada fará. Uma ativação representa um momento temporal em que um método é chamado e pode desencadear mais de um método.</p>
<p>Message1</p> 	<p>Mensagem Utilizado para interligar as ativações entre os objetos, indicando o nome do método que está sendo executado.</p>
<p>Message2</p> 	<p>Retorno Representa o retorno que o método devolve, indicando apenas um boolean (True/False) ou então um dataset (conjunto de dados).</p>

Abaixo temos um exemplo do diagrama de seqüência do sistema de restaurante:



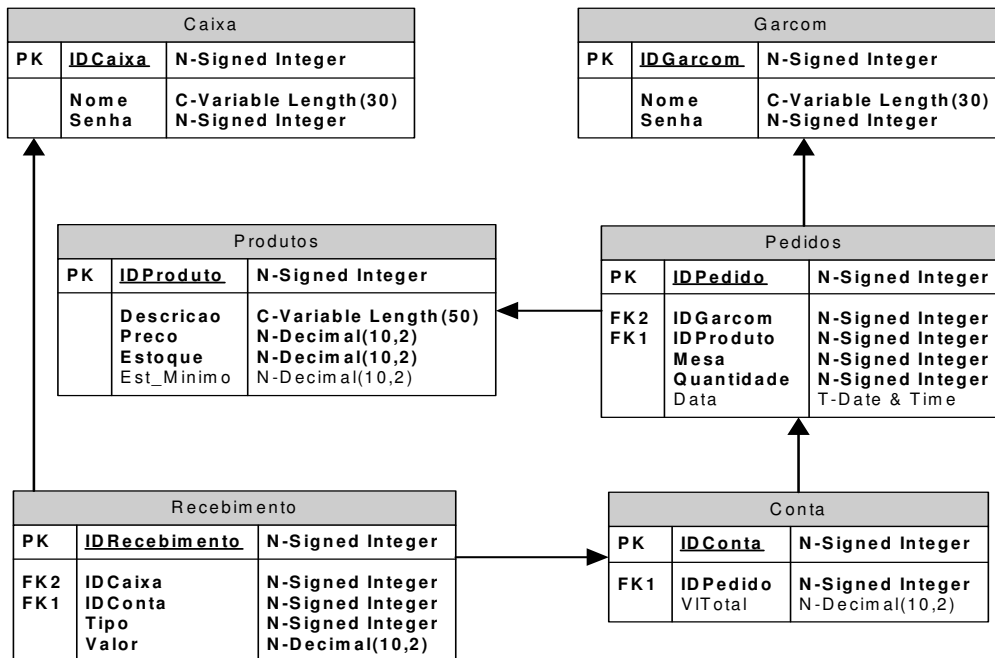
4.2.5 Diagrama de Dados

O diagrama de dados gerado neste momento já identifica o nome das tabelas e das colunas. Este modelo se aproxima muito do modelo fisicamente implantado, com a diferença básica de que o tipo de dados é IDL ou também chamado de “portável”.

Para entender o diagrama siga a tabela abaixo:

Símbolo	Descrição
Setas de interligação	Chave Estrangeira (FK) Identifica que uma tabela depende de outra. Por exemplo, o código do garçom na tabela de pedidos exige que este exista na tabela de garçom. Chaves estrangeiras valida a tabela por coibir o cadastro em uma tabela filho caso na tabela pai não exista o correspondente.

PK	Chave Primária Serve como identificador único em cada tabela. Por exemplo, no caso dos produtos geramos um código (IDProduto) que não pode ser repetido e ficar em branco. Em uma pessoa, o RG seria uma PK
Colunas em negrito	Obrigatória (not null) Define que a coluna não pode estar sem preenchimento. Poderia estar com dados em branco, ou seja, o usuário pode digitar espaços em branco, mas é obrigatório colocar algo.



No exemplo acima utilizamos dados portáveis, uma vez que no modelo lógico apenas sugerimos os servidores e produtos que serão utilizados, mas ainda não foram definitivamente escolhidos. O tipo de dados será automaticamente migrado no momento que for definido o produto, utilizando o tipo de dados mais próximo do definido no modelo lógico.

4.2.6 Design de Alto Nível

Chamado de High-Level Design, é um protótipo de como deverá ser o layout das tela de input de dados. Não define em linguagem, mas define basicamente as telas permitindo que no desenvolvimento sejam seguidas estas regras. Por exemplo, podemos definir o tamanho da tela, os tipos de controles gráficos utilizados, a cor e logotipos, etc.

Segue abaixo o modelo de design da tela de vendas que o garçom utilizará:

Este modelo foi elaborado levando-se em conta que nas tecnologias candidatas especificamos que o equipamento do garçom seria um Pocket PC ou um Palm, por isso as telas são menores e com todos os dados contidos para evitar navegação.

Não há detalhes como por exemplo, os labels, data e hora real na statusbar, nome dos objetos, etc. Mas há informações suficientes para conceituar o layout que o sistema deverá adotar.

4.3 Otimização

A otimização é como os processos de refinamento já analisados anteriormente. Nesta fase do lógico podemos verificar vários itens:

Item verificado	Descrição
Redundância	Verifique se o mesmo dado não foi colocado em duas tabelas, componentes ou objetos. Muitas vezes isto pode acontecer ao estar se definindo um grande projeto. Notamos também isto nas interfaces gráficas, muitas vezes duplicadas durante o planejamento.
Especialização	Cada componente deve ter uma função distinta. Deve-se evitar componentes com funções diferentes agrupados, conceito chamado de coesão, caso eles possam ser chamados por meios diferentes ou não tem relação direta entre si.
Escopo	O componente tem que cumprir todas as exigências do projeto, não contendo mais ou menos funcionalidades do que foi efetivamente contratado pelo cliente.
Correlação	Os relacionamentos tanto na base de dados quanto dos objetos deve estar correto. Algumas vezes podemos criar relacionamentos que não são sempre obrigatórios. Por exemplo, dizer que para cada conta temos apenas um recebimento, uma vez que alguns clientes podem dividir a conta entre si, exigindo o registro de diversos recebimentos para uma mesma conta.
Tipo	Neste item verificamos se as chamadas são síncronas ou assíncronas. Síncronas são funções que devem acontecer em uma ordem lógica, como por exemplo, a impressão do pedido na cozinha sempre ocorre na seqüência da inserção de um novo pedido pelo garçom. Por outro lado a função de fechamento de conta não acontece em decorrência da soma dos pedidos, uma vez que o cliente pode pedir apenas uma prévia de sua conta. Funções assíncronas servem para indicar operações que podem acontecer em outro momento. Por exemplo, um DOC bancário pode ser feito a qualquer hora do dia, mas ele só é processado a meia-noite do dia em que foi registrado. Este processo é muitas vezes chamado também de batch.
Controle	É importante conhecer os conceitos de coesão e acoplamento. Coesão é quando uma operação está tão intrinsecamente à outra que é agrupada em um único componente. Por exemplo, todas as vezes que eu apagar um funcionário pai eu automaticamente apago seus filhos, e apenas apago filhos quando o pai é apagado, o que gera uma alta coesão. Outro importante conceito é acoplamento, que indica relacionamento entre componentes. No nosso sistema de exemplo temos acoplamento entre o componente pedido e cozinha, uma vez que todas as vezes que incluo um pedido devo imprimir o pedido. Este exemplo é tão acoplado que poderia em uma situação real ser coeso.

Ao terminar esta fase chegamos ao marco do final da fase de planejamento lógico e iniciamos o planejamento físico.

5 Planejamento – Modelo Físico

5.1 Introdução

O modelo físico é formulado por pessoal altamente técnico pois entra nos meandros do desenvolvimento do software. Apesar de ainda não ser o desenvolvimento, o modelo físico determina como os códigos devem ser escrito, nomes das colunas e tabelas, tamanho das colunas, nomenclatura dos componentes, regras de linguagem, componentização em camadas, etc.

5.2 Pesquisa

Na fase de pesquisa partimos com base no modelo lógico e nos UCs para definir algumas características técnicas, essenciais para o projeto. No modelo lógico foram definidas as tecnologias candidatas, e agora escolheremos a que mais se encaixa, utilizando alguns itens exemplificados abaixo:

Item	Requisito
Performance	Definir o tempo de resposta que o software deve assumir como mínimo, por exemplo, em uma aplicação web a navegação entre páginas não pode ultrapassar 5 segundos.
Facilidade de uso	Qual é a tecnologia com mais profissionais habilitados e levando-se em conta a curva de aprendizado se teremos tempo hábil para a solução
Custo benefício	Os gastos com treinamento e suporte compensa a implementação de uma plataforma mais barata?
Instalação	Qual é o grau de dificuldade de instalação e entrega da tecnologia. Algumas são extremamente simples de serem usadas, mas no momento da implantação é necessário copiar dezenas de arquivos.
Suporte	O suporte da tecnologia em seu ambiente está habilitado ou seus profissionais não possuem conhecimento caso ocorram problemas.
Reutilização	Qual das tecnologias candidatas permitirá crescimento com reaproveitamento do que for feito. Alguns modelos de tecnologias são muito rígidos, não aceitando o acesso por outras plataformas e sistemas. Por exemplo, hoje componentes baseados em XML deve ser reconhecido em qualquer produto.
Segurança	A tecnologia possui realmente a segurança integrada ao software, ou a segurança é apenas proprietária, ocasionando falta de conhecimento interno.

Estes são exemplos de como escolher o produto final da aplicação. Com estes parâmetros pesquisados e conhecidos passamos a fase de análise. O documento gerado neste momento segue como o exemplo abaixo:

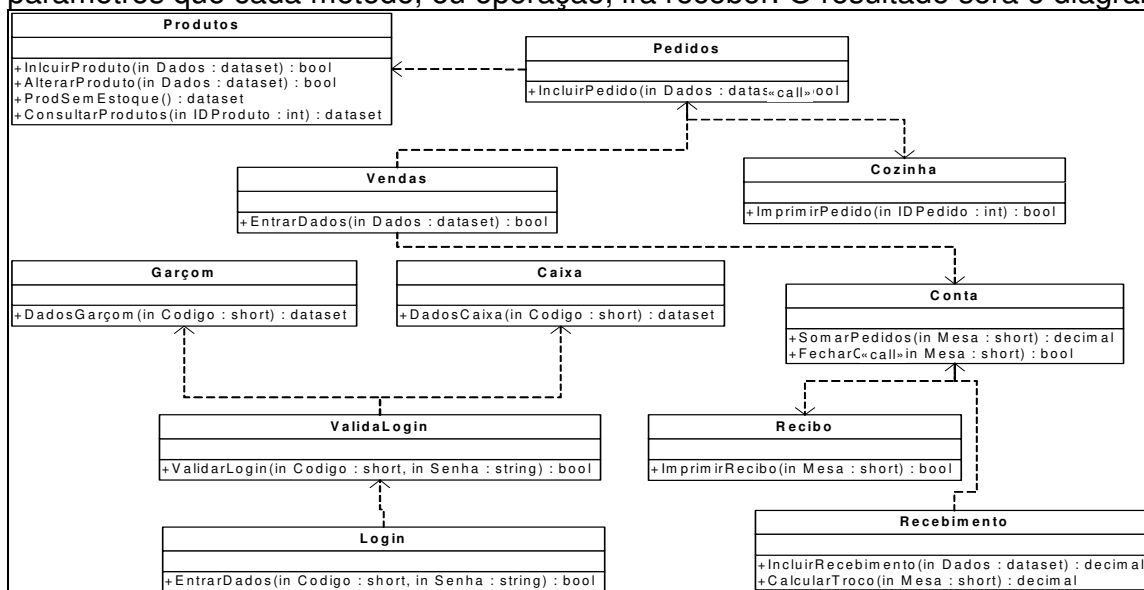
Requisito	Definição
Sistema operacional	Windows Server 2003 Standard Edition
Linguagem de programação	C#
Ferramenta RAD	Microsoft Visual Studio .NET 2003
Banco de dados	Microsoft SQL Server 2000
Equipamento para atendimento	HP iPAQ 9530 com Wi-fi e Pocket PC 2003
Equipamento de caixa	Pentium III com RAM de 256 MB e HD de 20 GB
Segurança	-Active Directory para acesso a rede -Autenticação e autorização de sistema por software
Tempo de resposta	O tempo de transação não deve ultrapassar 5 segundos
Apresentação	-Windows Forms em .NET Compact Framework nos Pocket PC -Windows Forms em .NET Framework nos caixas
Arquitetura da aplicação	Layered-Stateless-Client-Server
Servidor de Aplicação	COM+ no Windows 2003

5.3 Análise

Alguns diagramas e dados são importantes e revisados neste momento.

5.3.1 Diagramas de Classe e Seqüência Revisados

Primeiramente analisamos os diagramas de classe, database e seqüência para verificar e adaptá-los ao produto e tecnologia escolhida. Definiremos detalhe nas classes anteriormente definidas em UML. No diagrama anterior de classes possuíamos o nome dos métodos, mas não especificamos parâmetros de entrada e de saída. Neste momento iremos atualizar o modelo inserindo os parâmetros que cada método, ou operação, irá receber. O resultado será o diagrama abaixo:

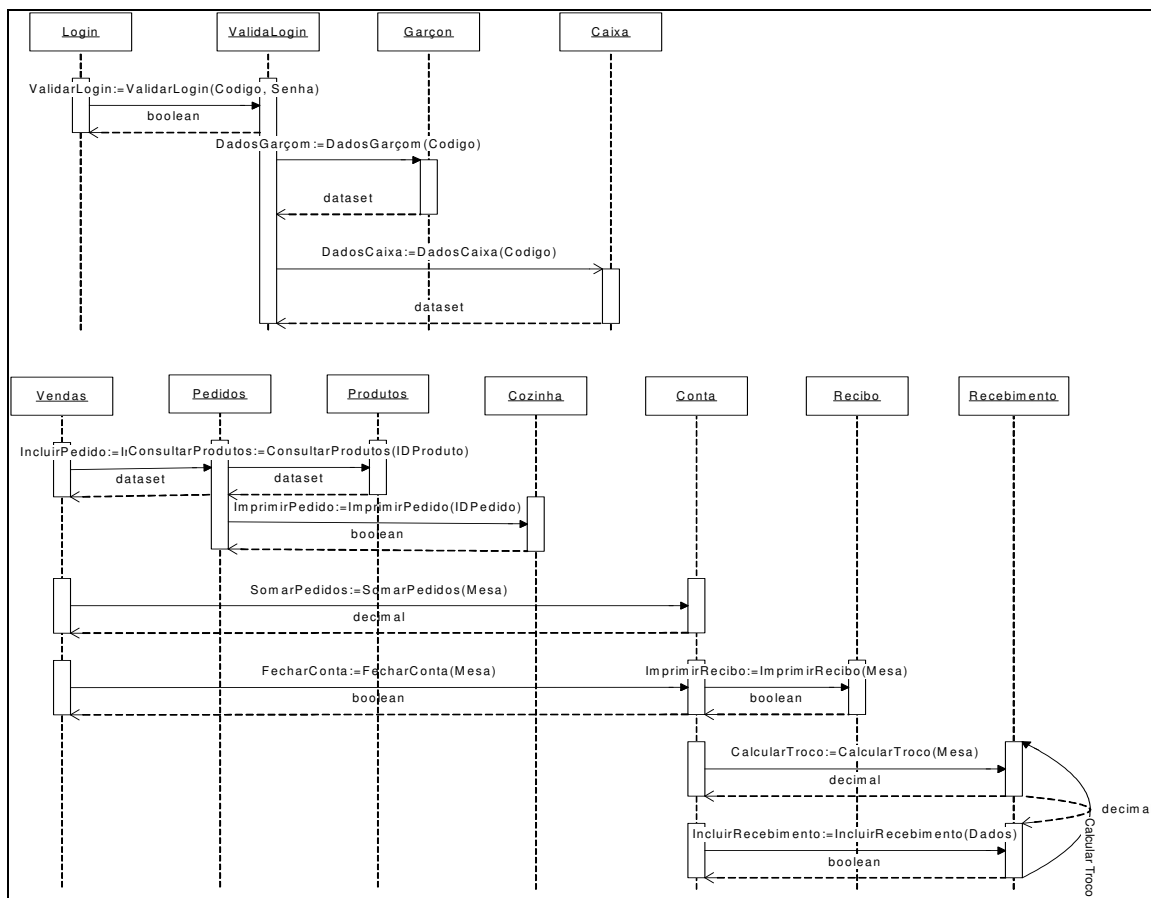


Note que temos os parâmetros de entrada baseados nos tipos de dados do C#. Também foi alterado o nome das operações. Os parâmetros entre parêntesis são os parâmetros de entrada, aqueles enviados ao método. O parâmetro que aparece após os parênteses e dois pontos é o retorno do método.

Portanto, definimos para a função EntrarDados no componente Login que ele receberá o código e a senha, onde o código é um inteiro curto e a senha uma seqüência de caracteres. O retorno deste método será um booleano, indicando true ou false.

Como pode ser visto na redefinição das classes, foi incluído uma nova operação nos produtos, denominada ConsultarProdutos. Esse processo é espiral, uma vez que podemos pedir a quem definiu o modelo lógico para validar esta alteração.

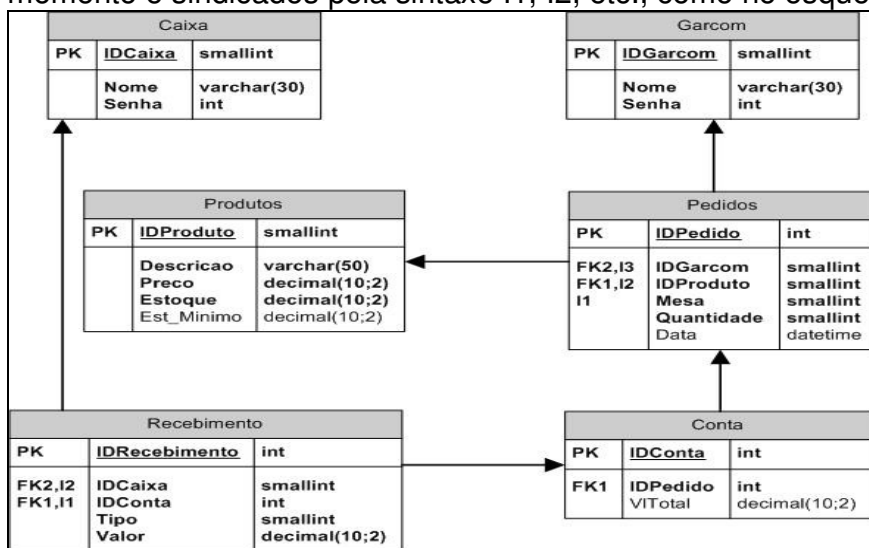
A alteração aprovada pelo analista do modelo lógico permitirá atualizar o diagrama de seqüência, que agora conta com mais um objeto e operação, além de incluir os parâmetros de entrada e saída, bem como os tipos de dados dos parâmetros:



5.3.2 Diagrama de Database Revisada

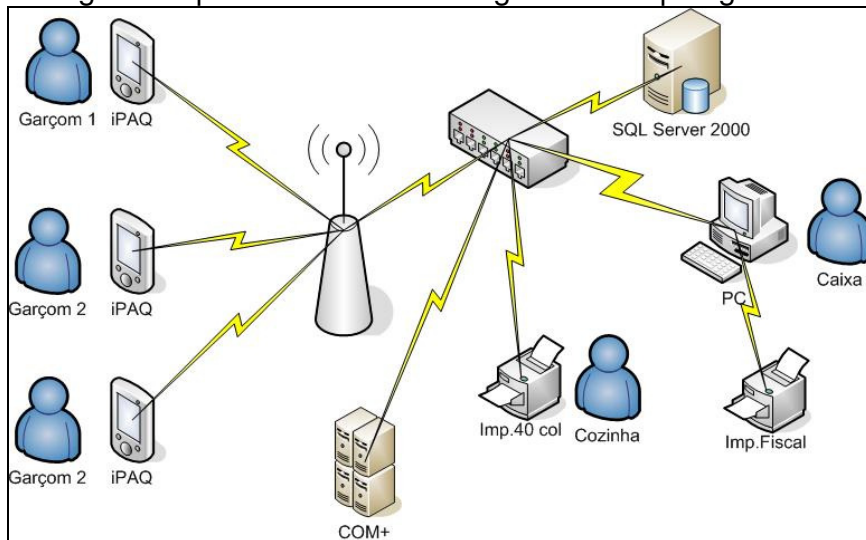
Revisar o database envolve apenas passar do modo portátil para o modo físico, ou seja passar de IDL para o tipo de dados que cada banco de dados possui.

O mesmo acontecerá com chaves primárias, estrangeiras e índices, sendo este ultimo criados este momento e sindicados pela sintaxe I1, I2, etc., como no esquema abaixo:



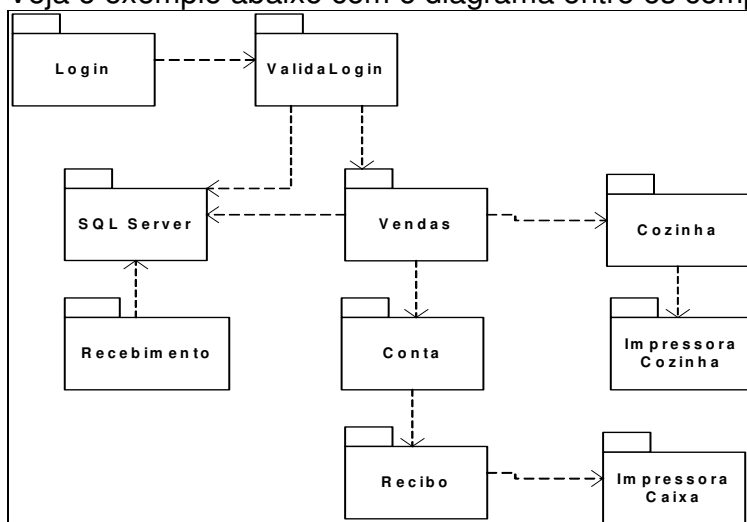
5.3.3 Diagramas de Topologia e Componentização

Após a revisão dos modelos anteriores podemos definir os próximos três diagramas a serem entregues. O primeiro deles é o diagrama de topologia futura:



Note que com este diagrama podemos exemplificar rapidamente os equipamentos, componentes de rede e usuários que estarão interagindo com o sistema.

Já o modelo de componentização servirá para indicar quantos componentes físicos (sejam estes formulários, dll ou executáveis) serão utilizados e qual o relacionamento entre eles. O modelo de componentes é totalmente baseado no diagrama de classes criado na fase de modelagem lógico. Veja o exemplo abaixo com o diagrama entre os componentes:

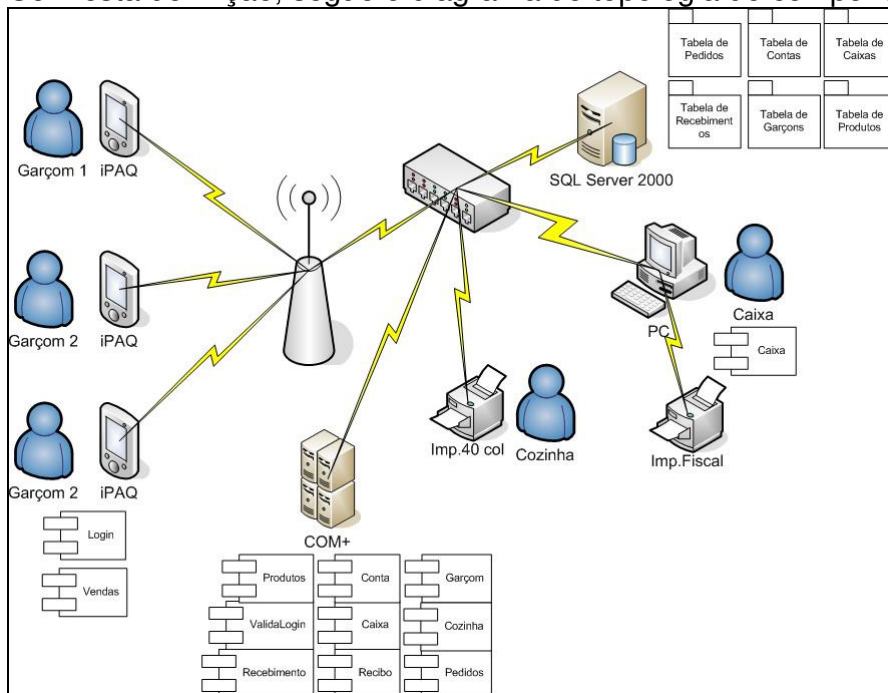


A partir do modelo de topologia e do modelo de componentização poderemos definir o diagrama de topologia de dados e componentes. Este indica em que diferente componente físico estará situado os componentes e dados.

Como em nossa definição fomos orientados a utilizar o modelo layered-stateless-cliente-server, sabemos que os clientes irão arquivar dados localmente durante uma operação, mas que deverão se comunicar constantemente com o servidor de banco de dados. Também entendemos pelo layered

que s componentes estarão em um servidor de componentes baseados no COM+, com processamento centralizado.

Com esta definição, segue o diagrama de topologia de componentes e dados:



5.4 Racionalização

O processo de racionalização pode ser definido como a fase onde os modelos anteriores são repensados. Não iremos na racionalização mudar as classes e objetos, mas sim pensamos em distribuição e empacotamento.

Por exemplo, até o momento temos uma classe chamada recibo que imprime um boleto sendo que esta operação sempre é executada ao ser fechada uma conta. Portanto, podemos dizer que estamos lidando com alto acoplamento e portanto estas duas classes podem estar compiladas fisicamente em um único assembleie.

Para que fique claro estes conceitos e como racionalizar, precisamos entender muito bem sobre o tipo de plataforma, gerenciadores de aplicações e linguagem que está sendo utilizada, já que as características que alistaremos agora mudam de uma para outra.

Conceito	Opções	Descrição
Gerenciamento de estado	Cliente	Esta opção é muito utilizada quando a aplicação será utilizada em soluções que o usuário final não pode contar com uma conexão fixa. Podemos exemplificar com os leitores de água utilizados atualmente pelos operadores. Estes precisam ter um arquivo em memória local no coletor, uma vez que no final do dia descarregam na base de dados de cobranças. Não seria possível manter o coletor conectado.
	SQL	Utilizado em ambientes altamente conectados, onde todos os dados estão centralizados no servidor SQL Server, Oracle, Sysbase ou outro produto. O desenvolvimento da aplicação é mais simples e alterações refletem instantaneamente.
	Cookie	Quando o cliente é web temos o problema comum da perda de dados cada vez que o usuário abre e fecha o browser. Para persistir dados no cliente podemos usar cookies que são arquivos texto com variáveis, e ler estes arquivos em páginas. O problema dos cookies existe porque alguns browsers e políticas de segurança proíbem este recurso por expor dados do usuário.

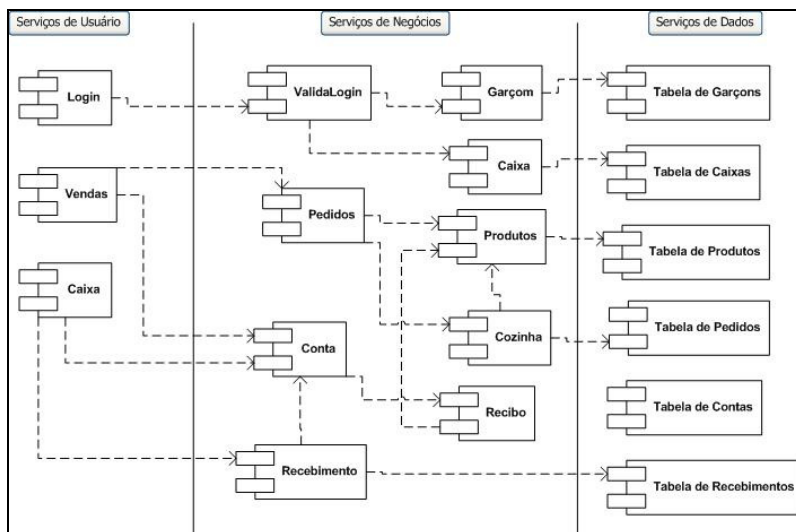
	Application State	Os dados são arquivados no servidor IIS ou COM+ de forma pública. Muito útil para guardar variáveis que são comuns a todos os usuários da aplicação e que são voláteis, como por exemplo, cotação do dólar. Se guardarmos a cotação em DBMS teremos o problema do alto consumo de rede, enquanto fazer a leitura uma única vez no dia e guardar na memória do servidor de aplicação reduz este tráfego de rede.
	Session State	Este acontece quando o usuário se conecta pelo browser. Cada diferente conexão via browser é uma diferente sessão. O problema neste modelo é que o usuário tem um período de timeout baixo, o que exige rapidez nas navegações para que a sessão não expire por inatividade.
Design	Escalabilidade	Envolve criar as chamadas "farms" ou "clusters". Esse recurso é colocar cada parte da sua aplicação em um diferente servidor. Sistemas não desenvolvidos em camadas não podem ser separados, portanto no dia em que um servidor ficar lento terá que ser trocado. Já em aplicações onde os componentes são separados, podemos chegar ao ponto de colocar cada componente em um servidor de aplicação diferente.
	Performance	Tenha bem em mente que tempo de resposta pode invalidar a aplicação. Imagine um sistema de telemarketing onde o acesso aos dados do cliente demore 40 segundos. O cliente que está pagando a ligação irá se irritar com o atendente, gerando um clima desfavorável para sua aplicação.
	Gerenciamento	Criar uma aplicação sem pensar na manutenção é um erro comum. Se a aplicação for distribuída comercialmente ou em grande escala isto fica mais complicado. Uma aplicação monolítica instalada em 3000 máquinas, para ser atualizada irá criar um verdadeiro problema. Portanto, ao pensar em modelo de componentes, a centralização em um servidor de aplicação torna a manutenção simples.
	Reutilização	Algumas partes de um componente podem ser acessadas por mais de uma diferente interface, e nestes casos a separação deste componente em pedaços menores melhoraria caso apenas uma das interfaces tenha que sofrer alterações, e ela é compartilhada entre aplicativos.
	Contexto	Funções que envolvem contabilidade não tem qualquer relação com as operações de vendas, portanto a separação entre elas ajuda na manutenção.
	Granularidade	Este conceito é muito importante. Alta granularidade envolve cada método ou poucos métodos por componente, enquanto baixa granularidade envolve cada componente conter todos os métodos relativos a função executada. Por exemplo, em nosso componente produtos temos quatro métodos, portanto estamos com baixa granularização, mas caso coloquemos cada um dos quatro métodos em um diferente componente estaríamos em alta granularidade.

Por fim, considere o que já foi citado anteriormente chamado de alta coesão e acoplamento. A alta coesão é quando o processo é chamado de atômico. Um processo atômico é aquele que acontece em um único momento, seja este temporal, funcional ou relacional. Por exemplo, podemos dizer que a impressão do recibo é atômica ao fechamento da conta, pois não podemos emitir o recibo sem fechar a conta primeiro. Em contrapartida não podemos dizer que o processo de vendas seja atômico com fechar conta, uma vez que vendas não apenas faz isto, mas também insere pedido, portanto é de baixa coesão.

Quanto a acoplamento é a dependência entre componentes. Quando um componente está separado de outro mas ele obrigatoriamente precisa deste, chamamos de alto acoplamento. Já componentes que são independentes são de baixo acoplamento.

Muitas vezes se misturam os termos coesão e acoplamento. Para ficar claro, coesão é o fato de diferentes métodos interagirem entre si, independente se estão em um mesmo componente ou não. Já o conceito de acoplamento envolve o componente estar ou não no mesmo assembleie.

Levando em conta os conceitos agora levantados, vamos colocar no diagrama de distribuição o modelo de nossa aplicação:



Terminados estes modelos, temos a aplicação documentada em UML.

5.5 Modelo de Implementação e Programação

Nesta ultima fase definimos como deverá ser feita a codificação do sistema. Muito dirigida aos programadores, exige alto nível técnico e bom experiência com desenvolvimento.

A função do modelo de programação é com que todos os programadores entendam os comandos e implementações, como por exemplo, nome de objetos, tipo de comentários, escopo de funções, tipo de dados, etc.

Segue abaixo alguns dos muitos tópicos que podem ser explorados neste modelo:

Item	Descrição
Tecnologia	Qual o tipo de recurso da tecnologia escolhida será utilizado, como por exemplo, APIs do sistema operacional, controle transacional do COM+, número de pacotes que serão gerados, se utilizaremos multi-thread, formulários MDI ou SDI, drivers para acesso a dados, etc.
Estado	As classes serão implementadas em statefull (cheio) ou stateless (vazio). Stateless é quando o componente recebe na lista de parâmetros do método todos os dados que necessita. Statefull são aplicações onde é necessário primeiramente informar os parâmetros um por um e apenas depois de todos os parâmetros informados executa-se a operação que não recebe parâmetros. Em nosso exemplo utilizamos sempre statefull, pois é mais simples de ser utilizado e não necessita que o componente guarde dados em memória, uma vez que ele recebe o que precisa e devolve tudo o que sabe, "esquecendo" os valores que lhe foram informados.
Chamada	Assemblies podem ser in-process ou out-process. Todas as dlls são in-process, ou seja, precisam que algum programa as chame para que rodem, elas por si só não podem ser executadas de forma autônoma. Por outro lado, executáveis são out-process, eles não precisam que alguém os chame, eles por si só têm esta capacidade de execução autônoma. Normalmente utilizamos um único executável (chamado de host) que instância, ou chama, os componentes compilados como dlls.
Tratamento de erro	É comum nas empresas existirem tratamentos de erro padrão, exigindo que as mensagens e código de erro sejam padronizados entre os sistemas. Também se define como e onde os tratamentos devem ser executados e disparados.
Segurança	Define se os dados deverão estar criptografados e como isto deve ser feito, se a autenticação se fará por aplicação ou pelo sistema operacional, se haverá autorização, ou seja, o acesso a cada funcionalidade será independente, entre outros.
Distribuição	Seja local ou em rede, em que diretório os componentes dll e executáveis deverão estar copiados, assim como o nome que os diretórios receberão.
Escopo de métodos	Métodos podem ser públicos (visto por qualquer aplicação), privados (apenas vistos internamente) ou shared (visto apenas nos componentes da aplicação). Os métodos de nosso sistema, podem ou não ser executados por outros? Se positivo deverão ser públicos, se negativo shared ou privados.
Regras de validação	As regras de validação dos dados podem ser executadas dentro dos componentes ou diretamente nos bancos de dados por utilizar Stored Procedures, Triggers e Constraint.
Design	Apesar do modelo gráfico já ter sido fornecido, precisa-se definir o tipo e tamanho da fonte, objetos que deverão ser

gráfico	usados (grid, botão, tab, listbox, etc), tipos de formulários, etc.
----------------	---

Outros tópicos já foram abordados em otimização do modelo lógico, onde definimos modo assíncrono e síncrono e outros.

5.6 Otimização

Como a fases de racionalização anteriores, na otimização é feita uma revisão de todo o modelo. Esta revisão terá que ser considerada final, já que ao terminar o planejamento entrega-se aos programadores as classes com as assinaturas já codificadas na linguagem escolhida. Chamamos estes componentes de “stub”, uma vez que eles só contem o cabeçalho, lista de parâmetros de entrada e saída das classes e dos métodos. O código de implementação é então escrito pelos programadores.

Cada método, parâmetro e propriedade que os componentes e tabelas recebem são chamados de interface. Interfaces devem ser vistas como imutáveis, uma vez que, por exemplo, alterar uma função que tem três parâmetros para quatro parâmetros irá causar erro nos sistemas que utilizam apenas três parâmetros. O mesmo acontece com um database, não é possível alterar colunas, para fazer isso os gerenciadores de banco criam uma tabela nova e importam os dados. Então é bom lembra-se de que interfaces devem ser bem definidas na fase de planejamento para evitar remendos posteriores.

No caso do diagrama de database as ferramentas Visio, Erwin entre outras já gera os scripts de criação dos objetos, o que apenas é entregue ao DBA que criará um novo banco de dados e executará o script, incluindo índices e outras considerações de performance que entenda ser importante.

6 Desenvolvimento

6.1 Introdução

Erroneamente se diz que esta é a principal fase do projeto. Na realidade de projetos bem sucedidos esta fase é a mais demorada, mas não a mais importante. Se os planejamentos foram bem feitos, haverá muito pouca margem de erro.

Veja por exemplo o caso das classes “stubs” que a fase de planejamento irá gerar em C#:

```
public class Conta()
{
    public Conta()
    {
    }
    public decimal SomarPedidos(short Mesa)
    {
    }
    public bool FecharConta(short Mesa)
    {
    }
}
```

Note que a classe está montada. Cabe ao programador agora implementar conforme as definições os código que retornam dados. Levando-se em conta que no planejamento já consta como deve ser feito, temos garantia de que o sistema irá cumprir seu propósito.

Os programadores recebem também os diagramas de seqüência, bem como os diagramas de atividades e UCs. Com base no diagrama de atividades, como o mostrado anteriormente no planejamento conceitual, será possível entender claramente o que deve ser feito nos códigos internos do método. O diagrama de relacionamento, dependência e seqüência mostram qual componente chama outro, quando e o que ele deve informar.

É possível que neste momento os programadores discordem de certos passos, mas deve levar em conta que se o planejamento foi bem feito não haverá muita margem para alterações nesta altura do projeto. Por outro lado, levando-se em conta que o processo é espiral, caso realmente haja inconsistências, esta deve ser comunicada de forma apropriada e após se levantar os dados, fazer a correção caso seja necessário.

6.2 Provas de Conceito

Entende-se por prova de conceito reuniões e versões do sistema, principalmente protótipos. Imagine nossa aplicação de exemplo e lembre-se de que ela utiliza Pocket PC, um equipamento reconhecidamente limitado quando comparado a um PC. Neste ponto as provas de conceito realizam a importante tarefa de garantir que o equipamento irá suportar a aplicação.

Para a prova de conceito não é necessário estar todo o sistema pronto, nem mesmo estar desenvolvido o sistema especificado, uma vez que se pode utilizar outros exemplos mais rápidos e simples a fim de demonstrar o entendimento correto.

As provas de conceito servem principalmente para garantir que o pessoal de desenvolvimento tenha conhecimento da tecnologia e entendam a sua aplicação.

Por exemplo, se formos utilizar Internet podemos desenhar um site com duas ou três páginas de navegação para o cliente, usuário final e analistas dêem aval no modelo gráfico, posicionamento e tecnologia utilizada.

6.3 Versões Internas (internal releases)

Versões internas são similares as provas de conceito em objetivo, mas diferente no que demonstram. Nas provas de conceito o exemplo utilizado não necessita ser a solução proposta, mas apenas um exercício com a tecnologia.

Já as versões internas serão criadas periódica e sistematicamente. Algumas empresas como a Microsoft utilizam o conceito de "daily build" no final do expediente para no período noturno serem feitos testes com hardware, compatibilidade e bugs.

Estas versões também podem ser criadas ao final de cada implementação, como por exemplo, ao terminar a tela de login de nosso sistema e com as tabelas de garçons e caixas montadas podemos criar uma versão para que os analistas e testadores possam verificar por meio de testes pré-definidos a aplicação naquele componente específico.

Este passo pode parecer trabalhoso, mas fornece uma ferramenta extremamente útil por permitir que erros sejam tratados e resolvidos antes que se tornem partes de um sistema completo. É muito mais fácil corrigir erros em componentes individuais no início do desenvolvimento do que fazer isso com o sistema pronto, uma vez que pode ocasionar erros em cascata.

Alem dos testes de uso e funcionalidade as versões internas também possibilitam o processo de "code review". Este processo é normalmente executado por um programador mais experiente que não estará testando o sistema, mas sim a forma como foi escrito. Por exemplo, se foram feitos comentários descritivos, se os escopos estão corretos, layout conforme as especificações, etc.

Para muitos programadores o code review é um processo desnecessário, mas quando se trabalha em grandes equipes o code review garante que qualquer programador entenderá o código em caso de manutenção posterior.

É evidente esta vantagem quando imaginamos que um erro ou alteração pode ocorrer em um sistema anos após a sua implantação. Por isso, o code review ajudará que no futuro ao ser aberto o código fonte não haverá surpresas.

7 Estabilização / Testes

7.1 Introdução

A fase de estabilização e testes tem por objetivo utilizar métodos de testes elaborados para definir se a aplicação segue o modelo fornecido, não apenas tecnicamente, mas também do ponto de vista do usuário.

Nem sempre todos os erros são corrigidos. Alguns erros são postergados ou documentados.

Podemos exemplificar com um fogão doméstico. Ao se ligar o forno com a tampa de vidro abaixada o calor do forno poderá quebrar o vidro. Como este problema é de difícil solução por ser custosa, optou-se por colocar etiquetas de advertência.

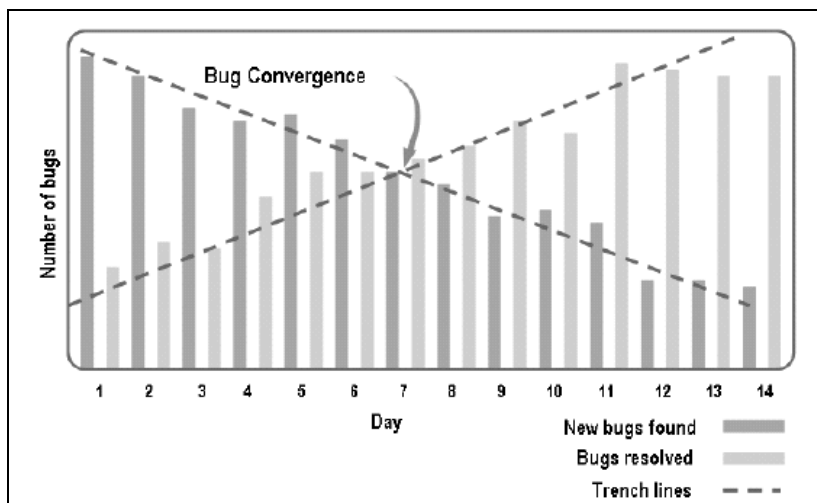
Do mesmo modo, alguns erros não podem ser corrigidos e passam a ser catalogados, como por exemplo, a distancia em que o Pocket utilizado pelo vendedor pode estar da antena antes da conexão cair.

Os testes a serem efetuados envolvem:

Teste	Itens verificados
Funcional	Utilizando os diagramas de atividade simulamos o usuário operando o sistema. Este teste é feito com diversas variações onde tentamos provocar erros e inconsistências.
Configuração	Utilizando-se diferentes hardwares e situações termais, distancias ou outras variações se o sistema continua funcionando, como por exemplo, afastar-se da antena do wi-fi.
Stress	Pode ser feito com simuladores para verificar a carga de dados que o sistema agüenta.
Performance	Verifica se a performance não é afetada por diversos fatores, como os testes relacionados acima.
Documentação e Help	Utilizando-se do manual do usuário e do help averigua se estes não estão pulando etapas ou desatualizados, além de esclarecedores o suficiente para o usuário final.
Paralelo	Quando um sistema é migrado simulamos uma operação no sistema antigo e no novo e o resultado não deve ser divergente para garantir que funcionalidades ou dados não estão sendo perdidos.
Regressão	Ao receber a correção de um erro anterior muitas vezes pode-se ter afetado outros componentes. O teste de regressão envolve testar funcionalidades relacionadas.

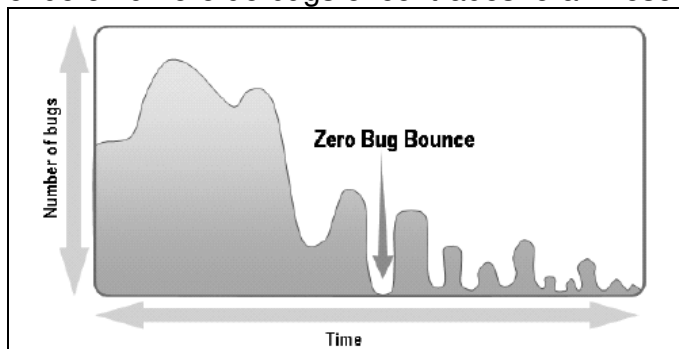
7.2 Convergência de Bugs e Zero Bounce

Entende-se por convergência de bug o momento em que o numero de erros ativos é menor do que o numero de erros corrigidos. Esta curva abaixo demonstra este conceito:



Pode-se dizer que este momento é importante porque demonstra que a equipe de desenvolvimento está trabalhando eficientemente. Este é o momento, ou marco, em que geramos uma versão Alpha. Versões alpha são restritas a um grupo de usuários pré-escolhidos que irão iniciar testes em ambiente real, se possível.

Após a convergência de bugs esperamos ocorrer o marco chamado de “zero bounce”, momento onde o numero de bugs encontrados foram resolvidos, como mostra o diagrama abaixo:



Este marco serve para o lançamento da versão candidata (release candidate).

7.3 Versões Candidatas

Versão candidata não quer dizer que todos os erros foram resolvidos ou documentados, mas que as funções do sistema estão estáveis e funcionando.

A versão candidata é importante porque nunca um teste em ambiente controlado simula o ambiente real do usuário final.

É necessário ao criar versões candidatas estar atento e receber rapidamente, com prazo pré-definido, as opiniões e erros relato pelos usuários testando.

Caso não seja um método ágil de relatório, o usuário irá simplesmente parar os testes ou então ignorar o erro e esperar que outra pessoa o faça.

8 Entrega

8.1 Introdução

Chegamos a versão final do produto e agora iremos para o deployment, ou entrega, deste. Muitos sistemas ótimos esbarram em dificuldades ao serem instalados. Por exemplo, todos queremos ter ar condicionado, mas muitos prédios de apartamento não são fabricados com suportes, o que exige além da compra do ar condicionado a contratação de um pedreiro, materiais e uma boa sujeira.

Para minimizar estes problemas podemos criar métodos que minimizem ou então garantam uma boa entrega e implantação de nossa solução.

8.2 Implantação de Tecnologias

O primeiro passo é implantar as tecnologias básicas, o que inclui sistema operacional, criação de diretórios e cópia de arquivos do sistema, bem como banco de dados e sistemas.

Esse passo é mais complicado quando estamos migrando sistemas existentes e neste caso sempre é prudente executar backup antes da alteração, principalmente alterações no banco de dados.

Para garantir crie um plano piloto, talvez em uma filial menor ou parte dos usuário seguindo algumas regras abaixo:

Grupo	Item	Descrição
Preparação	Objetivo	Tenha documentado, aprovado e tornado público porque o teste está sendo realizado e quais os problemas potenciais que podem ocorrer.
	Treinamento	Treine os usuários do piloto de forma exemplar e garanta que todos entenderam bem, uma vez que o piloto falhando, a solução naufragou.
	Suporte	Crie um agendamento e deixe com os usuários a lista de pessoas e plantões.
	Comunicação	Deixe de fácil acesso a comunicação com o suporte, crie planos de geração de relatórios.
	Contingência	Tenha um plano testado para permitir sistemas paralelos caso o piloto tenha que sofrer paradas para correções.
	Retorno	Não é desejável mas melhor do que gerar prejuízos. Admita caso não esteja apto a continuar a implantação e tenha um plano bem definido para retornar um backup, por exemplo, para o sistema antigo voltar a funcionar.
	Prazos	Prazos curtos mas que tenham ocorrido todas as situações funcionais é o ideal.
Implementação		Faça teste de falhas propositais no piloto. Documente os resultados da falha. Tente até mesmo gerar falhas críticas, pois irá testar como afetará o negócio e os risco serão melhor entendidos após isso.
Avaliação	Relatório	Forneça um site ou um contato por email para os usuários enviarem suas opiniões, de preferência anônimas.
	Pesquisa formal	Esta servirá para o pessoal de cargos mais altos e os diretamente envolvidos em decisões para definir o nível de satisfação com o novo sistema.
	Contagem de suporte	Verifique quais os problemas mais comuns que geraram suporte e como foram resolvidos. Problemas comuns são erros de documentação ou de sistema.

Muito cuidado com o chamado “quiet period” (período silencioso) onde os usuários se acostumaram com a aplicação mas ainda não utilizaram funções esporádicas, como por exemplo, relatórios mensais ou fechamentos.

Muitas vezes o dia a dia parece estar em ordem, mas o fechamento mensal provoca perda de dados ou dados inexatos, que podem até mesmo levar a empresa a dificuldades.

8.3 Transferência de Comando

Tendo conseguido chegar na excelência esperada, é hora de transferir comando.

No modelo Microsoft, denominado MSF, nós abrangemos da concepção (visão) a entrega do produto. Mas a operação dos sistemas após este estar implantado é definida por outros modelos

padronizados de mercado como o ITIL do qual a Microsoft gerou um modelo chamado MOF, Microsoft Operational Framework.

A transferência de comando deve ser feita com muito treinamento e demonstrações, com planos de contingência e mitigação bem documentados.

Ao transferir uma operação com sucesso garantirá que seus esforços em um novo projeto não serão interpelados por um projeto anterior.

Dedique tempo a transferência, não a leve como algo de menos importância, já que você poderá ter ganhado um verdadeiro filho por fazer uma má transferência.

Outro fator importante, é que se na transferência ocorrerem problemas, todo o seu trabalho anterior pode ser denegrido e levar o rótulo de incompetência.